



Product:	CDP2Qt
Product version:	1.2
Document ID:	UM - CDP2Qt
Doc revision:	A
Date:	16.12.2008
Pages:	20

Industrial Control Design AS



CDP2Qt 1.2 User Manual



Contents

1. INTRODUCTION.....	4
1.1. About.....	4
1.1.1. About this document.....	4
1.1.2. What is CDP2Qt?.....	4
1.1.3. What can be gained by using CDP2Qt?.....	4
1.2. Requirements.....	4
1.2.1. Tools and programs.....	4
1.2.2. Programming environment.....	4
1.2.3. Runtime libraries.....	4
1.3. Comparison of Design Concepts.....	5
1.3.1. Similar definitions.....	5
1.3.2. CDP Signals vs Qt Signals.....	5
1.3.3. CDP Connector and Qt Connections.....	5
2. INSTALLATION AND CONFIGURATION.....	6
2.1. About.....	6
2.1.1. About this chapter.....	6
2.2. Installation Guide.....	6
2.2.1. Download and install CDP.....	6
2.2.2. Download and Install Qt/Qtopia.....	6
2.2.3. Download and Install CDP2Qt.....	7
2.2.4. Test and run the CDP2Qt project.....	7
3. THE CDP DESIGNER WIDGETS.....	8
3.1. About.....	8
3.1.1. About this chapter.....	8
3.1.2. The Qt Designer widget box.....	8
3.2. The CDP Widgets.....	8
3.2.1. The CDPTutorWidget.....	8
3.2.2. The CDPLcdNumber.....	9
3.2.3. The CDPSliderWidget.....	10
3.2.4. The CDPSignalWidget.....	10
3.2.5. The CDPCconnectorWidget.....	12
4. MAKING CDP2QT APPLICATIONS.....	13
4.1. About.....	13
4.1.1. About this chapter.....	13
4.2. The different steps in the making of a CDP2Qt Application.....	13
4.2.1. Decide where to add functionality.....	13
4.2.2. Creating custom widgets and working with the Qt Designer.....	13
4.2.3. Adding the designer ui file to a CDP project.....	13
4.2.4. Running the application.....	14

5. EXTENDING THE CDP2QT INTERFACE.....	15
5.1. About.....	15
5.1.1. About this chapter.....	15
5.2. Creating CDP based Widgets	15
5.2.1. The CDPWidget class.....	15
5.2.2. Creating a CDP based widget.....	15
5.3. Extending the CDP2Qt Interface.....	16
5.3.1. Alternatives.....	16
5.3.2. Extending the CDPWidgets library.....	16
5.3.3. Creating a new CDPWidgets library.....	17
5.4. Extending the CDPDesignerPlugin.....	18
5.4.1. Alternatives.....	18
5.4.2. Extending the CDP designer plugin.....	18
5.4.3. Creating new plugins for the Qt Designer.....	20

1. Introduction

1.1. About

1.1.1. About this document

This document will describe how to install, extend and build applications using the CDP2Qt interface.

1.1.2. What is CDP2Qt?

The CDP2Qt Interface package includes a CDP2Qt template project, a CDP2Qt interface library, and a Qt designer plugin containing several CDPWidgets. The latter also includes a project file for building a static library of the widgets. In short, developers will create their projects using the Qt designer and dynamically load their ui files by the means of a CDP application that is linked with CDP2Qt. Users that wish to include their own custom plugins can easily extend the plugins. The framework will be explained in detail in the following chapters.

The most important feature of the CDP2Qt interface is thus that simple applications can be made without coding a single line of code. As the number of plugins grow, less coding is required.

1.1.3. What can be gained by using CDP2Qt?

The CDP2Qt interface will allow Qt developers to integrate their applications with a component based, real time and distributed control system, and make it possible to interface against a range of communication protocols. For users that are already familiar with CDP, Qt will present the possibility for making cross-platform GUI, and introduce a range of powerful features like database and direct-x support.

1.2. Requirements

1.2.1. Tools and programs

Before you can start building an application with CDP2Qt you must ensure that both CDP and Qt are installed on your system. In addition, you need to have a valid CDP license key in the license folder of the CDP root directory. Otherwise your executable will output error messages and not run properly.

1.2.2. Programming environment

The CDP2Qt interface runs on both Windows and Linux.

1.2.3. Runtime libraries

In contrast to the previously released QTServers, the CDP2Qt interface is built using dynamic C/C++ runtime libraries. Thus, you will need to download a non-standard version of CDP, since the libraries that are shipped with the standard package require static runtime libraries. Mixing libraries that are built with different runtimes will result in a massive list of link errors.

1.3. Comparison of Design Concepts

1.3.1. Similar definitions

Both CDP and Qt define a component based framework that makes programming fast and efficient. They also share some of the same definitions, like signals and connections. The following sections will briefly describe a few important differences. Read the CDP and Qt documentation for more details.

1.3.2. CDP Signals vs Qt Signals

The Qt Signal is a special kind of function that can be emitted by an event such as the push of a button. By connecting the Qt Signal to a function defined as a slot, that function will be run when the signal is emitted. See the Qt documentation at trolltech.org for a more detailed description.

In CDP, a Signal is also a means of communication, but here the signal might as well be connected to a component in a different application, running on a different physical machine. The CDP signals are handled and updated by the CDP core components, and though they are objects, they can be used within the application as if they were ordinary variables. There are no slots, as signals are defined as output and inputs, and are routed to each other by name. This is configured in XML.

1.3.3. CDP Connector and Qt Connections

CDP Connectors are used to send messages to other CDP Components. Unlike the connection between Qt Signals and Qt Slots, these components do not have to reside on the same physical machine. The address of the target (remote) CDP Component is specified in XML.

2. Installation and Configuration

2.1. About

2.1.1. About this chapter

This chapter explains how to install and configure CDP2Qt.

2.2. Installation Guide

2.2.1. Download and install CDP

Download the CDP installer from the ICD homepage at www.icd.no. Users that do not already have a valid CDP license can get one for free. Visit the “free trial” page to get an evaluation license. Note that the CDP2Qt interface requires CDP libraries built with dynamic C/C++ runtimes. Get them at the download page and copy them into the Libs directory at the CDP install path (\$CDPBase).

2.2.2. Download and Install Qt/Qtopia

Download Qt or Qtopia from the Trolltech homepage at www.trolltech.com. Users that download the open source version of Qt must build the source files themselves. See the following sub sections for step by step instructions. Note that these steps are intended for Microsoft Visual Studio users only. The steps might also be different for different versions of Qt.

Set up the environment

Start out by adding a few environment variables to your system. In Windows XP, this is done by right clicking My Computer and selecting Properties. Under the Advanced tab, click the button named Environment Variables. The first variable to add is QMAKESPEC. This variable provides QT with information about your system. Give it one of the following values, depending on your visual studio version.

- Visual Studio 6.0 is represented by win32-msvc
- Visual Studio .NET (2003) is represented by win32-msvc.net
- Visual Studio 2005 is represented by win32-msvc2005

Also add a variable named QTDIR with the path to your QT directory, C:\Qt\4.4.0 . Finally, it is time to add QT to the environment PATH. This is done by adding C:\Qt\4.4.0\bin; to the beginning of the value. Check that the paths are added correctly by typing `echo %VariableName%` in the command prompt.

Building the source

Before you start configuring the source, you should check that the Visual Studio compiler files are in the path. This is, in most cases, ensured by opening the command prompt from the Visual Studio program group:

- Start > Microsoft Visual Studio 200x > Visual Studio Tools > Visual Studio 200x Command Prompt
- Check that the environment is set up in a correct matter by typing `nmake /?`

Browse to the folder where the Qt source was unzipped ,and type the following to list the available build options:

- C:\Qt\4.4.0> `configure -help`

Run configure with the options of your choice (answer yes to all questions):

- `C:\Qt\4.4.0> configure -debug-and-release -fast`

When the configuration is finished, you may build QT by typing:

- `C:\Qt\4.4.0> nmake`

2.2.3. Download and Install CDP2Qt

Download the CDP2Qt interface from the ICD homepage at www.icd.no. Run the installer to place the CDP2Qt files into the \$CDPBase directory. The CDPBase environment variable holds the path to the CDP installation directory. Thus, you can always get to this directory by typing `cd %CDPBase%` in Windows command prompt, or by typing `cd $CDPBase` in Linux bash. The following steps explain what to do next.

Configure and build the CDP2Qt project in Linux and Windows (required to build the plugin)

- Open command prompt or bin bash with the environment set up as as described above. Then execute one of the following lines depending on the OS you are running.
- Windows: `cd %CDPBase%/CDP2Qt && qmake && nmake && nmake release`
- Linux: `cd $CDPBase/CDP2Qt && qmake && make && make release`

Configure and install the CDP plugin for the Qt Designer in Windows (Qt open source)

- Start > Microsoft Visual Studio 200x > Visual Studio Tools > Visual Studio 200x Command Prompt
- Check that the environment is set up with the correct variables and versions. As described in the previous section, this can be carried out by typing `echo` followed by an environment variable enclosed in percent signs (e.g. `echo %variable%`). The QTDIR variable should print “C:\Qt\4.4.0”, QMAKESPEC should print “win32-msvc.net” and PATH must include “C:\Qt\4.4.0\bin”.
- The CDP2Qt project files uses %CDPBase_Develop% in addition to %CDPBase%. Set the variable as described in the previous section or by typing `SET CDPBase_Develop=%CDPBase%`. In Linux, this would typically be added to .bashrc, or by typing `export CDPBase_Develop=$CDPBase` in a bash shell.
- `cd %CDPBase%\CDP2Qt\CDPDesignerPlugin && qmake && nmake && nmake install`

Configure and install the CDP plugin for the Qt Designer in Windows (Qt commercial)

- Start > Qt by Trolltech v4.4.0 > Qt 4.4.0 Command Prompt
- Now, follow the same steps as described for the open source version.

Configure and install the CDP plugin for the Qt Designer in Linux

- Linux is no different from Windows. Just replace %VariableName% with \$VariableName and you can follow the above instructions. In short, that is as follows.
- Open a bash window and check that the environment includes the right variables (using `echo` command).
- `cd $CDPBase/CDP2Qt/CDPDesignerPlugin && qmake && make && make install`

When the above steps has been carried out successfully, the Qt Designer should include several CDP widgets in its toolbox (the next time it is started). Use these widgets along with standard widgets or extend the CDP2Qt interface with additional plugins to use custom widgets in the Qt Designer. The template project in \$CDPBase/CDP2Qt/CDP2Qt shows how to use the interface to dynamically load ui files.

2.2.4. Test and run the CDP2Qt project

The CDP2Qt project is extremely simple to use and extend. Use the Qt Designer (now containing CDP plugins) to edit the ui file specified in %CDPBase%\CDP2Qt\CDP2Qt\Application\Components\CDP2Qt.xml, and run the GUI by executing the batch files or CDP executable (Linux) located in the Application directory. Note that the executable must be run as root in Linux. See the following chapters for more details.

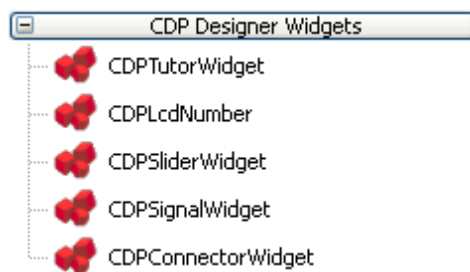
3. The CDP Designer Widgets

3.1. About

3.1.1. About this chapter

This chapter describes the various CDP Widgets that are included in the CDP2Qt plugin for the Qt Designer.

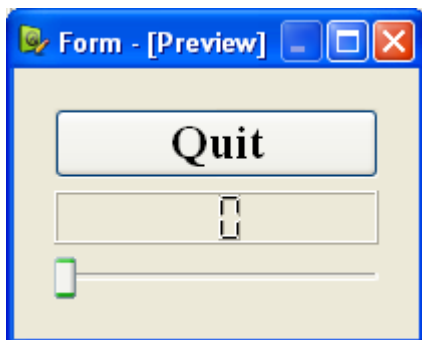
3.1.2. The Qt Designer widget box



The above widgets should be listed in the Qt Designer widget box if the CDP designer plugin has been successfully installed. Users are allowed to add their own custom widgets to the list using the framework described in this document. The following section describes the various CDP Widgets.

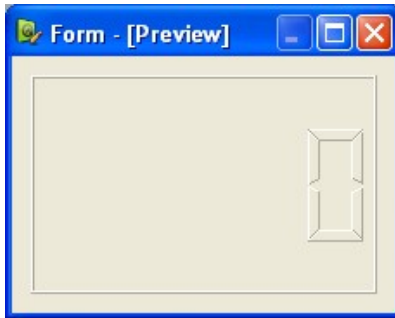
3.2. The CDP Widgets

3.2.1. The CDPTutorWidget

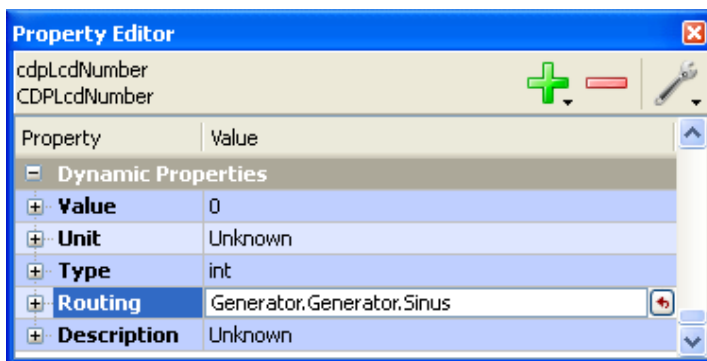


The CDPTutorWidget doesn't contain any CDP functionality at all. It is just a simple widget with a slider that controls an LCD and a button to quit the application. The only reason for the widget to be a part of the CDP2Qt package is to demonstrate how to add widgets with functionality (and also to show how the example from the previous QTSERVER interface can be implemented in CDP2Qt).

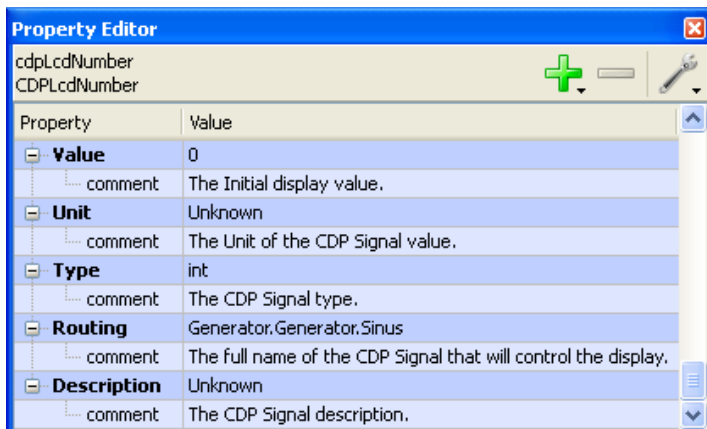
3.2.2. The CDPLcdNumber



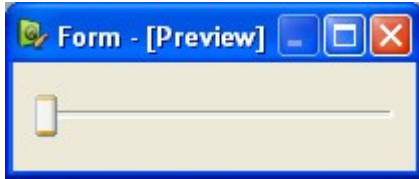
The CDPLcdNumber looks identical to the standard LCD Number widget. The difference is that this widget can show any CDP Signal on its display. Signal routing can be set by selecting the widget and editing its dynamic properties. This is illustrated in the below image.



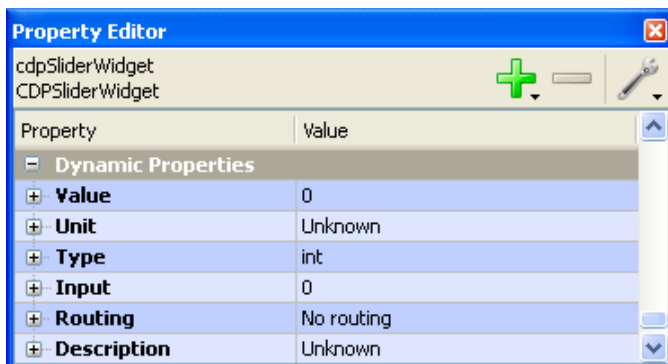
Help comments can be viewed by clicking the plus sign in front of the property names. This feature is similar for all the CDP widgets discussed in this document.



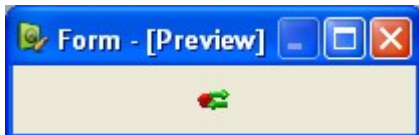
3.2.3. The CDPSliderWidget



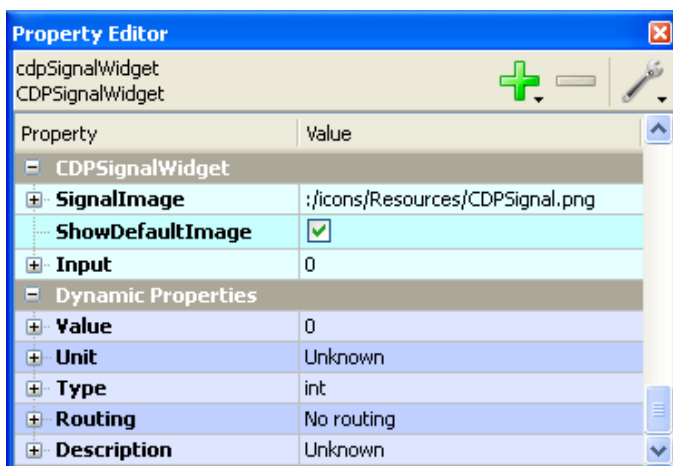
As with the CDPLcdNumber, the CDPSliderWidget looks identical to the Qt slider, but on the inside it includes CDP functionality. When the slider is dynamically created by the CDP2Qt interface, it will get connected to a CDP output signal which will get the same name as the slider object. The connection is valid for both directions. Thus, the slider will change position if the CDP output signal is changed by an external component. The dynamic properties are shown in the below image. Note that the slider direction (horizontal or vertical) is changed in the QAbstractSlider section of the properties.



3.2.4. The CDPSignalWidget

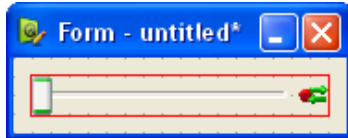


The CDPSignalWidget can be used to connect standard Qt widgets to any CDP input or output signals. In addition to the dynamic properties of standard CDP widgets, the CDPSignalWidget also includes properties for hiding and showing the signal image, and also options for changing the image altogether. It is even possible to drag and drop new images on the widget to avoid writing long file paths.

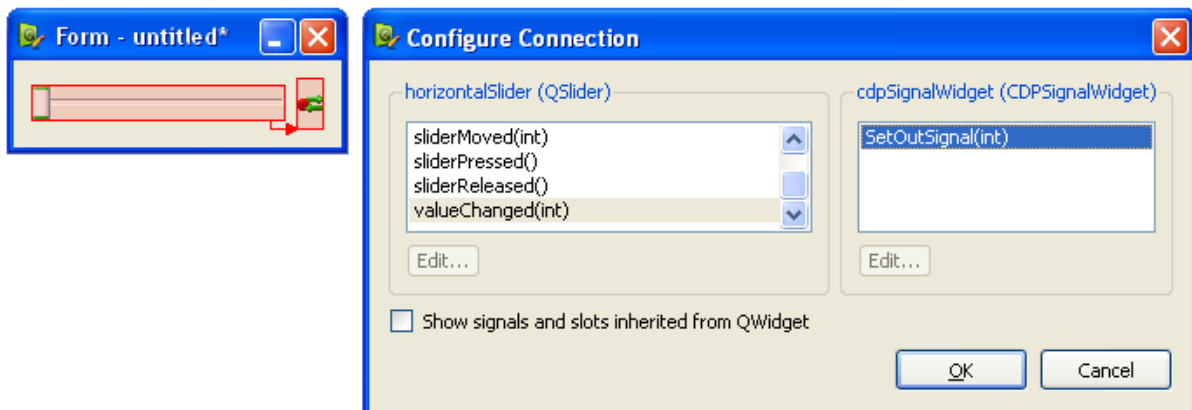


Adding a CDP signal to a standard Qt widget

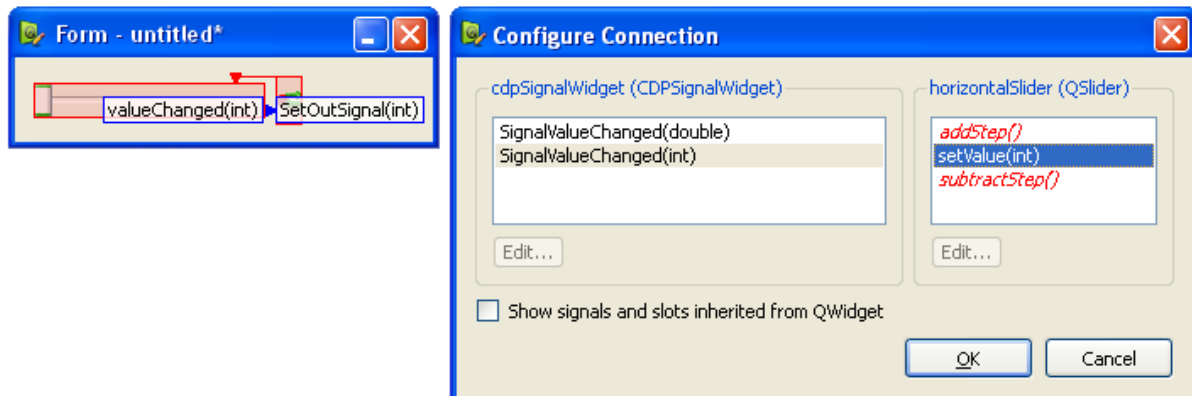
With the CDPSignalWidget it is thus possible to provide any standard Qt widget with CDP signals, without using the CDP2Qt framework. To illustrate this, we will show how to make a CDPSliderWidget in a few simple steps. Start out by dragging a Horizontal Slider and a CDPSignalWidget to a form, and place them in a Horizontal Layout. The form should then look something like the following.



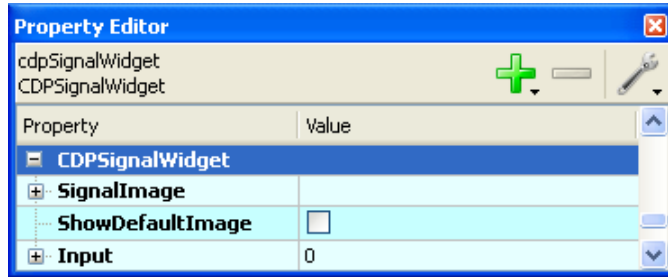
Hit the F4 button on your keyboard to edit signals and slots. Then click the Horizontal Slider and drag the mouse over to the CDPSignalWidget. When releasing the mouse, you will be presented with a list showing available signals and slots. Select the signal named valueChanged(int) and the slot named SetOutSignal(int). Press OK.



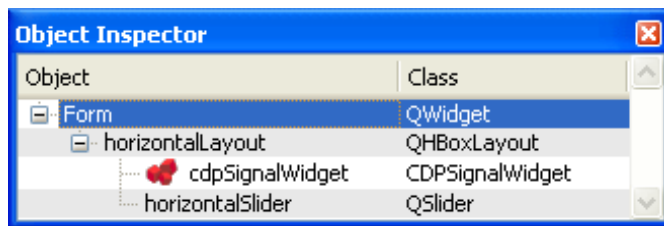
To make the slider move if the output signal is changed by an external application, like a CDPBrowser or Internet Explorer, we also need to make a connection the other way around.



Now, it is time for editing the properties. Hit the F3 button to go back into “edit widgets” mode, and select the CDPSignalWidget. Locate the CDPSignalWidget section in the properties menu, and disable the ShowDefaultImage property. This will hide the signal image and make the slider look just like an ordinary Horizontal Slider. Note that to prevent empty space within the layout, you must also select the Horizontal Layout and set the layoutSpacing to zero.



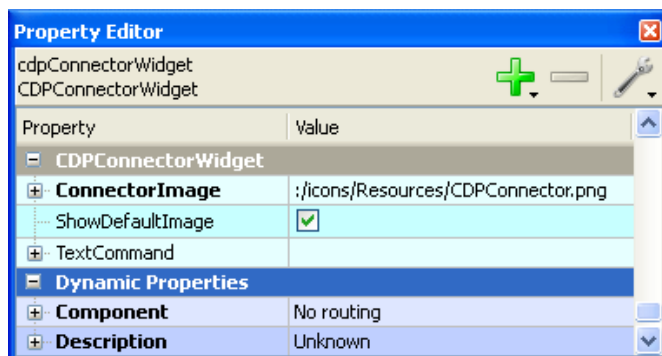
When the image of the CDPSignalWidget is hidden, it might be a bit hard to find the widget clicking with the mouse. The easiest way to locate hidden widget is to use the “Object Inspector” box and double-click the widget that you want to find. You may then edit its properties as usual.



3.2.5. The CDPCConnectorWidget



The CDPCConnectorWidget can be used to send text messages to any CDP component. Typical use is to connect the widget to Push Button using the same approach as with the CDPSignalWidget example. Then edit the properties with the text message to send, and the full name of the component to send it to.



4. Making CDP2Qt Applications

4.1. About

4.1.1. About this chapter

This chapter explains how to work with CDP2Qt using the Qt Designer and the template project located in path/to/install/CDP2Qt/CDP2Qt.

4.2. The different steps in the making of a CDP2Qt Application

4.2.1. Decide where to add functionality

Developers can choose to add most of their application within Qt widgets, or choose to keep their widgets really simple and implement most of the functionality inside CDP components. As a rule of thumb, time critical functionality should be inserted within CDP components as they are prioritized over CDP2Qt. Other than that, it is completely up to the programmer. Qt programmers should note that they can distribute work to CDP components which does not need to reside on the same physical machine.

4.2.2. Creating custom widgets and working with the Qt Designer

All custom widgets can be developed as usual following the instructions in the Qt documentation. To use these widgets with the Qt Designer, they must be implemented as designer plugins. The alternative is to use existing widgets as placeholders. Both solutions are described in the Qt documentation, but the former will be explained in detail in the following chapter. Note that if you want to implement CDP functionality in your custom widget, it needs to inherit the CDPWidget class.

Always keep in mind that all slots and signals that are going to connect with CDP signals or CDP connections need to be available to the top level widget. Widgets that are created in the constructor of another widget won't have any visible signal or slots to the Qt Designer, unless the top level widget implements such signals and slots explicitly. This will become clear as developers get familiar with the Qt Designer.

4.2.3. Adding the designer ui file to a CDP project

When the developer has finished creating and configuring the ui, it must be added to a CDP project that includes a CDP2Qt component. This project requires a special CDPMain.cpp file, since the Qt application needs to run in the main thread (also noted as the GUI thread in Qt terminology). The necessary files are included in the template project located in path/to/install/CDP2QtInterface/CDP2Qt. Feel free to copy this folder and add more components as needed. Note that CDP has lots of interfaces and components that can be added without writing a single line of code. Read the CDP documentation for more information.

After compiling the template project, the executable should be placed in the Application directory. The ui file can be placed in any directory as long as the correct path is added to the CDP2Qt.xml file, located in the application component directory. Note that the path must be relative to the CDP executable.

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- CDP2Qt interface component. -->
<Component Name="CDP2Qt" Type="CDP2Qt">
```

```
<!-- The Qt GUI refresh rate -->
<fs>10</fs>

<!-- The Qt ui page (path must be relative to executable) -->
<ui>test.ui</ui>

<InstanceHelp></InstanceHelp>
<HtmlPage></HtmlPage>
<Activate>1</Activate>

<Alarms>
</Alarms>

<Parameters>
</Parameters>

</Component>
```

4.2.4. Running the application

Since the application relies on dynamic libraries, the executable will only run if those libraries are present. In both Windows and Linux, the libraries have to be added to the library search path. Windows users can choose to edit a .bat file and use that file to set up the right environment. Alternatively, it is possible to copy the libs into the application directory or simply run the executable from the Qt command prompt. In Linux, the project can be compiled with the rpath option, or the library paths can be added to the LD_LIBRARY_PATH. Similar to Windows, the libraries can also be copied into the application directory.

5. Extending the CDP2Qt Interface

5.1. About

5.1.1. About this chapter

This chapter provides details about how to create widgets with CDP functionality, and how to extend both the CDP2Qt interface and the CDPDesignerPlugin.

5.2. Creating CDP based Widgets

5.2.1. The CDPWidget class

The CDPWidget class must be inherited by all widgets that include CDP functionality. When such widgets are created by the CDP2Qt interface, the widgets are added to a global list of CDP based widgets. All widgets in this list are configured by the interface based on dynamic properties in the ui file. The interface will also allocate any CDP signals and connections that the widget might have. Due to limitations in the XML capabilities of the Qt ui files, CDP2Qt can only handle one CDP signal and one CDP connector per widget. Further CDP signals and CDP connectors can be added by the means of CDP Signal Widgets and CDP Connector Widgets.

5.2.2. Creating a CDP based widget

Creating a CDP based widget only requires a few simple steps. In this section, these steps are explained using the CDPSignalWidget as an example. Note that the source code for all the CDPWidgets are located in path/to/install/CDP2QtInterface/CDPDesignerPlugin. The qmake project file within this directory will create makefiles for building the designer plugin, while the CDPWidgets directory contains a project file for building a static plugin library for CDP2Qt.

The class declaration of the CDPSignalWidget includes a special define, CDP2QT_WIDGET_EXPORT. This define is necessary when the class is going to be built as both a designer plugin and a plugin for CDP2Qt. Note that when we are writing about a plugin for CDP2Qt we are actually talking about a static library. The class implements a CDP signal, and thus we need to inherit CDPWidget. To get a nice frame around the widget we also inherit QFrame.

```
class CDP2QT_WIDGET_EXPORT CDPSignalWidget : public QFrame, public CDPWidget
```

Shortly after the Q_OBJECT and Q_PROPERTY macros, we find the constructor declaration. In addition to the well known QWidget* argument, we also find QString. Both are required to make the widget load properly.

```
CDPSignalWidget(QWidget *parent = 0, const QString &name= 0);
```

The next function to notice is UpdateGui(). This function is run periodically by the CDP2Qt interface at the frequency set by the component fs (CDP parameter). Within this function we would typically emit changed variables or set output signals and so on based on GUI values. When a widget requires updating at a different rate than provided by the CDP2Qt component fs, it is recommended to add a timer to the class instead of using this function.

```
virtual void UpdateGui(void);
```

Since the CDPSignalWidget is going to include a CDP signal, we need to add a protected SignalBase pointer. If we also wanted a CDP connector, we could simply add a CDPCConnector member variable (not a pointer).

```
SignalBase *m_pSignal;
```

The CDP signal needs to be added to the signal list inherited from the CDPWidget base class. Forgetting this part will make the signal pointer unavailable for the CDP2Qt interface and thus prevent it from being created. CDP signals and connectors must be added in the class constructor using the CDPWidget member lists: m_signalList and m_connectorList. Note that the signal list requires that the user appends a pointer to the signal pointer while the connector list requires a connector pointer. Also note the initializer list that calls CDPWidget with a object pointer and name. Forgetting this will prevent the widget from getting added to the global list of CDP based widgets.

```
CDPSignalWidget::CDPSignalWidget(QWidget *parent, const QString &name)
    : QFrame(parent), CDPWidget(this, name), m_pSignal(0),
      m_lastSignalInput(0), m_direction("0") //<< IMPORTANT!!!
{
    m_signalList.append(&m_pSignal); //<< IMPORTANT!!!

    layout.addWidget(&m_labelBox);
    layout.setAlignment(Qt::AlignVCenter | Qt::AlignHCenter);
    layout.setContentsMargins(0,0,0,0);
    setLayout(&layout);

    setAcceptDrops(true);
}

// The UpdateGui funtion as implemented for the CDPSignalWidget
void CDPSignalWidget::UpdateGui(void)
{
    if(m_lastSignalInput!=m_pSignal->GetDouble())
    {
        m_lastSignalInput=m_pSignal->GetDouble();
        emit SignalValueChanged(m_lastSignalInput);
        emit SignalValueChanged(static_cast<int>(m_lastSignalInput));
    }
}
```

5.3. Extending the CDP2Qt Interface

5.3.1. Alternatives

The CDP2Qt interface is extendable with an unlimited number of widgets. Users can choose to either add their widgets to the CDPWidgets library, or use this project as a template for making their own standalone library.

5.3.2. Extending the CDPWidgets library

Extending the CDPWidgets library is extremely simple. We will once again use the CDPSignalWidget as an example and show how to add this widget to the CDPWidgets project. First, add the CDPSignalWidget source files to the CDPWidgets project file. Then, open CDPDesignerPluginLibBuilder.cpp and add an include to CDPSignalWidget.h. Next, we add the the following lines to CDPDesignerPluginLibBuilder::CreateNewWidget:

```
else if(className == QLatin1String("CDPSignalWidget"))
    pWidget = new CDPSignalWidget(parent, name);
```

Rebuild the CDPWidgets project.

5.3.3. Creating a new CDPWidgets library

To create a new CDP2Qt widgets library, we need to make a new builder class to provide the CDP2Qt interface with knowledge about how to create our widgets. The class must inherit CDP2QtBuilder and include a CreateNewWidget function as illustrated in the following lines:

```
/**
(c)2008 ICD AS

CDPDesignerPluginLibBuilder header file.
*/

#ifndef CDPDESIGNERPLUGINLIBBUILDER_H_INCLUDED
#define CDPDESIGNERPLUGINLIBBUILDER_H_INCLUDED

#include <CDP2QtBuilder.h>

#include <QWidget>
#include <QString>

class CDPDesignerPluginLibBuilder : public CDP2QtBuilder
{
public:
    CDPDesignerPluginLibBuilder();
    virtual ~CDPDesignerPluginLibBuilder(){};

    virtual QWidget * CreateNewWidget(const QString &className, QWidget *parent = 0,
                                     const QString &name = QString());
};

#endif // CDPDESIGNERPLUGINLIBBUILDER_H_INCLUDED
```

The implementation file is shown here:

```
#include "CDPDesignerPluginLibBuilder.h"

#include "CDPSliderWidget.h"
#include "CDPConnectorWidget.h"
#include "CDPSignalWidget.h"
#include "CDPLcdNumber.h"
#include "CDPTutorWidget.h"

CDPDesignerPluginLibBuilder::CDPDesignerPluginLibBuilder()
: CDP2QtBuilder()
{
}

QWidget *CDPDesignerPluginLibBuilder::CreateNewWidget(const QString &className, QWidget *parent, const
QString &name)
{
    QWidget *pWidget = 0;

    if (className == QLatin1String("CDPConnectorWidget"))
        pWidget = new CDPConnectorWidget(parent, name);
    else if (className == QLatin1String("CDPSignalWidget"))
        pWidget = new CDPSignalWidget(parent, name);
    else if (className == QLatin1String("CDPLcdNumber"))
        pWidget = new CDPLcdNumber(parent, name);
    else if (className == QLatin1String("CDPTutorWidget"))
        pWidget = new CDPTutorWidget(parent);
    else if (className == QLatin1String("CDPSliderWidget"))
        pWidget = new CDPSliderWidget(parent, name);

    return pWidget;
}
```

//<< IMPORTANT TO REMEMBER!!

//<<INSERT YOUR WIDGETS HERE

**//<< NOTE THAT ONLY CDP BASED
//<< WIDGETS REQUIRE THE
//<< NAME PARAMETER**

Next, we need a header file for making an instance of the builder class.

```

/**
(c)2008 ICD AS

CDPDesignerPluginLib header file. Include this file in the project to use the library.
*/
#ifndef CDPDESIGNERPLUGINLIB_H
#define CDPDESIGNERPLUGINLIB_H

#include "CDPDesignerPluginLibBuilder.h"

/** Instantiate the ExamplePluginLibBuilder for this object */
CDPDesignerPluginLibBuilder cCDPDesignerPluginLibBuilder;

#endif //CDPDESIGNERPLUGINLIB_H

```

Now, all we have to do is to include the above file in the Libraries.h file of our CDP2Qt project. Note that you also need to make a qmake project file for building the library, and add the library as a dependency in the CDP2Qt project file.

5.4. Extending the CDPDesignerPlugin

5.4.1. Alternatives

When making ui files for the CDP2Qt framework, users can choose to create new plugins for the Qt Designer, extend the CDPDesignerPlugin, or to use an existing widget as a placeholder for their custom widget.

5.4.2. Extending the CDP designer plugin

Widgets that do not include CDP functionality can be added to the CDP designer plugin like any ordinary widget. Create an interface class that inherits QObject and QDesignerCustomWidgetInterface, and implement the various functions as described in the Qt documentation.

```

#ifndef CDPSIGNALWIDGETINTERFACE_H
#define CDPSIGNALWIDGETINTERFACE_H

#include <QDesignerCustomWidgetInterface>

class CDPSignalWidgetInterface : public QObject, public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
    CDPSignalWidgetInterface(QObject *parent = 0);

    bool isContainer() const;
    bool isInitialized() const;
    QIcon icon() const;
    QString domXml() const;
    QString group() const;
    QString includeFile() const;
    QString name() const;
    QString toolTip() const;
    QString whatsThis() const;
    QWidget *createWidget(QWidget *parent);
    void initialize(QDesignerFormEditorInterface *core);

private:
    bool initialized;
};
#endif //CDPSIGNALWIDGETINTERFACE_H

```

For widgets that include CDP signals, the `domXml()` function must include XML in the following style:

```
QString CDPSignalWidgetInterface::domXml() const
{
    return "<widget class=\"CDPSignalWidget\" name=\"cdpSignalWidget\">\n"
        "<property name=\"Value\" >\n"
        "  <string comment=\"The Initial display value.\">0</string>\n"
        "</property>\n"
        "<property name=\"Unit\" >\n"
        "  <string comment=\"The Unit of the CDP Signal value.\">Unknown</string>\n"
        "</property>\n"
        "<property name=\"Type\" >\n"
        "  <string comment=\"The CDP Signal type.\">int</string>\n"
        "</property>\n"
        "<property name=\"Input\" >\n"
        "  <string comment=\"Set to 1 for input or 0 for output.\">0</string>\n"
        "</property>\n"
        "<property name=\"Routing\" >\n"
        "  <string comment=\"The full name of the CDP Signal that will control the display.\">No
routing</string>\n"
        "</property>\n"
        "<property name=\"Description\" >\n"
        "  <string comment=\"The CDP Signal description.\">Unknown</string>\n"
        "</property>\n"
        "<property name=\"SignalImage\" >\n"
        "  <string comment=\"CDP Signal display image.\">:/icons/Resources/CDPSignal.png</string>\n"
        "</property>\n"
        "</widget>\n";
}
```

Widgets that include CDP connectors need to have a `domXml()` function with XML in the following style:

```
QString CDPCConnectorWidgetInterface::domXml() const
{
    return "<widget class=\"CDPCConnectorWidget\" name=\"cdpConnectorWidget\">\n"
        "<property name=\"Component\" >\n"
        "  <string comment=\"Remote component name.\">No routing</string>\n"
        "</property>\n"
        "<property name=\"Description\" >\n"
        "  <string comment=\"Short connector description.\">Unknown</string>\n"
        "</property>\n"
        "<property name=\"ConnectorImage\" >\n"
        "  <string comment=\"The display image.\">:/icons/Resources/CDPCConnector.png </string>\n"
        "</property>\n"
        "</widget>\n";
}
```

Finally, we just need to add the interface to `CDPDesignerPlugin.cpp`. Add an include to the interface header file, and append the interface to the widget variable member in the `CDPDesignerPlugin` constructor.

```
#include "CDPDesignerPlugin.h"

#include "CDPTutorWidgetInterface.h"
#include "CDPLcdNumberInterface.h"
#include "CDPSliderWidgetInterface.h"
#include "CDPSignalWidgetInterface.h"
#include "CDPCConnectorWidgetInterface.h"

CDPDesignerPlugin::CDPDesignerPlugin(QObject *parent)
    : QObject(parent)
{
    widgets.append(new CDPTutorWidgetInterface(this));
    widgets.append(new CDPLcdNumberInterface(this));
    widgets.append(new CDPSliderWidgetInterface(this));
    widgets.append(new CDPSignalWidgetInterface(this));
    widgets.append(new CDPCConnectorWidgetInterface(this));
}
```

```
}
```

```
QList<QDesignerCustomWidgetInterface*> CDPDesignerPlugin::customWidgets() const  
{  
    return widgets;  
}
```

```
Q_EXPORT_PLUGIN2(cdpdesignerplugin, CDPDesignerPlugin)
```

5.4.3. Creating new plugins for the Qt Designer

Use the project located in path/to/install/CDP2Qt/ExamplePlugin as a template, or see the Qt documentation for more details.