A Step-by-step Tutorial of Building a GUI in ROS with Qt / C++

Author & Maintainer: Peng Xu
Email: robotpengxu@gmail.com
Code: https://github.com/xpharry/ROSCppGUI/tree/master/QtROS_GUI
Update: April 22, 2017

1. Overview

A graphical user interface (GUI) enables the end users to interact with ROS through graphical icons and visual indicators.

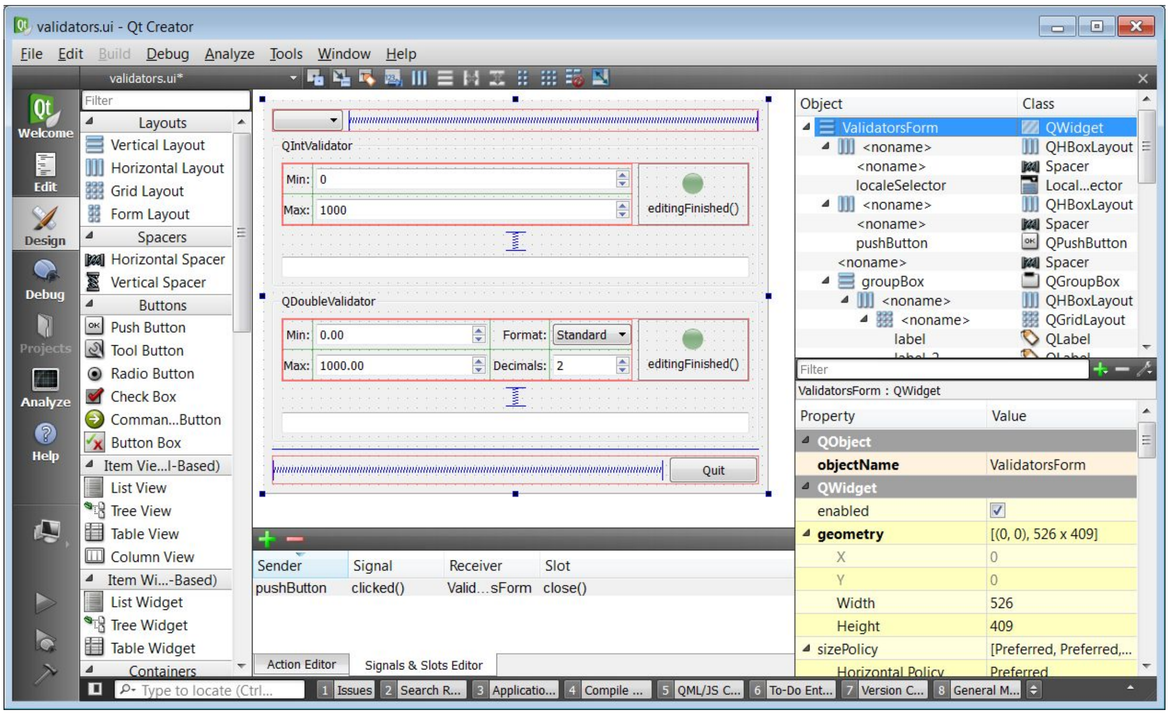
In fact, there has been multiple tools which can be used in Linux or Ubuntu to make a GUI, such as Qt, Gambas, GTK+ and Perl. Also in Github, there are a number of examples of ROS GUI, while most of them are written in Python. As C++ developers, we expect to find a generic way to build a GUI for ROS programs wrote in C++.

A lightweight and easy-to-build tool to write a GUI would be perfectly saving developers' time and energy and helps better focusing on the original technical topic, which is ROS here. Among those above, Qt highlights itself for its broad use in different platforms and plenty of open-sourced resources.

2. Set Up a Qt Development Environment

Qt is a cross-platform application framework that is widely used for developing application software that can be run on various software and hardware platforms with little or no change in the underlying codebase, while having the power and speed of native applications.

Qt provides a friendly GUI tool, Qt Creator, for you to design your own GUI. An example GUI created by Qt Creator is shown as following,



Sender	Signal	Receiver	Slot
pushButton	clicked()	Valid...sForm	close()

Object	Class
ValidatorsForm	QWidget
<noname>	QHBoxLayout
<noname>	Spacer
localeSelector	Local...ector
<noname>	QHBoxLayout
<noname>	Spacer
pushButton	QPushButton
<noname>	Spacer
groupBox	QGroupBox
<noname>	QHBoxLayout
<noname>	QGridLayout
label	QLabel

Property	Value
QObject	
objectName	ValidatorsForm
QWidget	
enabled	✓
geometry	[(0, 0), 526 x 409]
X	0
Y	0
Width	526
Height	409
sizePolicy	[Preferred, Preferred,...
Horizontal Policy	Preferred

We will highly rely on this tool in the future steps.

1) Install the Latest Qt SDK

The latest version of Qt is Qt 5.8 (download), though you can still follow an old version tutorial, which indicate a step by step path to install Qt SDK 4.8 and Qt SDK 5.0 on Ubuntu.

Once you are done with that, the Qt Creator tool can be found in the "tool" directory.

Alternately, you can use **apt-get** as below to install QtCreator if you are comfortable with that and you will get Qt Creator 4.0.2 based on Qt 5.7.0.

```
$ sudo apt-get install qtcreator
```

2) Build Your First Qt program

This tutorial, Create-Your-First-Qt-Program-on-Ubuntu-Linux, for those who are new to Qt, would definitely help. You are going to build a simple hello world program, but enough to be familiar with the widgets and the "qmake" use.

3) Install Useful Qt-ROS Packages

Fortunately, there are two ROS packages for building a ROS GUI template quickly using C++. These will be used in Step 3

```
$ sudo apt-get install ros-indigo-qt-create
$ sudo apt-get install ros-indigo-qt-build
```

4) Import an Existing ROS Project to QtCreator

Follow the steps as below,

- Click "File->Open File or Project"
- Choose "CMakeLists.txt" in ros package folder
- Choose or create a build directory for building and running

Note: Not all the files in the ROS package folder will be automatically imported to QtCreator, such as "include" folder. You might need to import them manually.

3. Create a Qt ROS GUI template

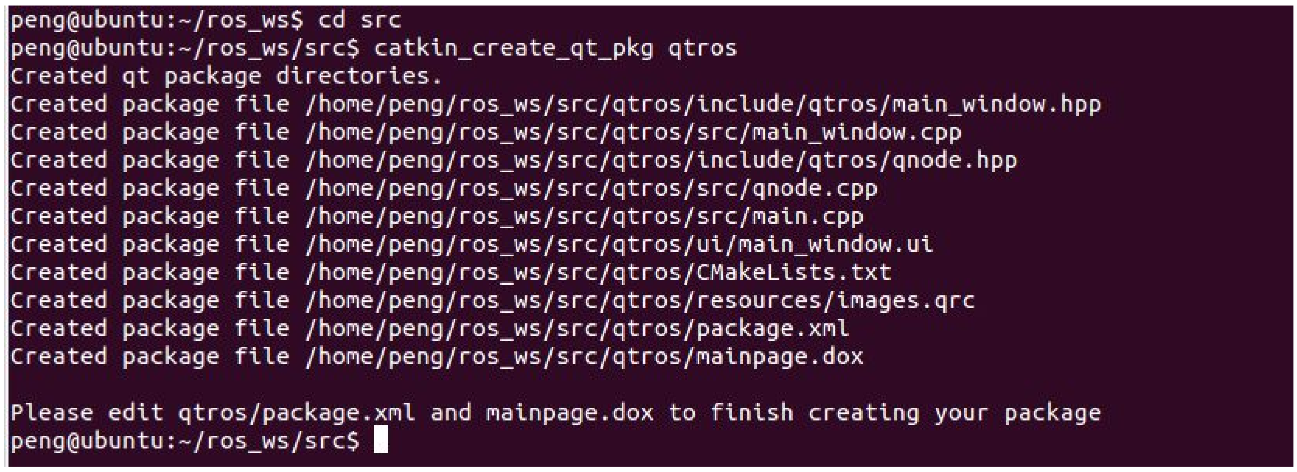
With the packages imported in Step 2.3, you can simply use one command, **catkin_create_qt_pkg**, to build a complete Qt ROS GUI template, which you can also refer to ROS Wiki.

Concretely, navigate to your ROS workspace, type the following commands in the terminal,

```
$ cd src
$ catkin_create_qt_create qtros
```

where "qtros" is the package name you created.

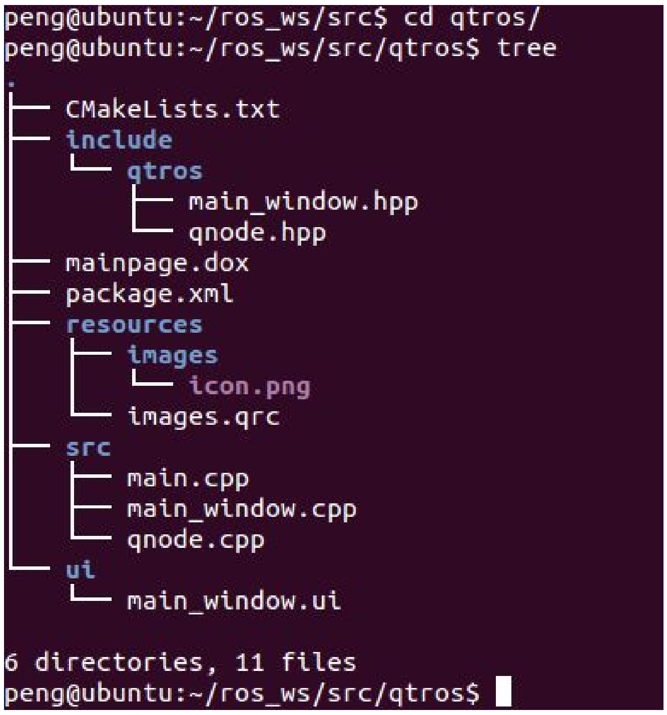
Normally, it will show like below,



```
peng@ubuntu:~/ros_ws$ cd src
peng@ubuntu:~/ros_ws/src$ catkin_create_qt_pkg qtros
Created qt package directories.
Created package file /home/peng/ros_ws/src/qtros/include/qtros/main_window.hpp
Created package file /home/peng/ros_ws/src/qtros/src/main_window.cpp
Created package file /home/peng/ros_ws/src/qtros/include/qtros/qnode.hpp
Created package file /home/peng/ros_ws/src/qtros/src/qnode.cpp
Created package file /home/peng/ros_ws/src/qtros/src/main.cpp
Created package file /home/peng/ros_ws/src/qtros/ui/main_window.ui
Created package file /home/peng/ros_ws/src/qtros/CMakeLists.txt
Created package file /home/peng/ros_ws/src/qtros/resources/images.qrc
Created package file /home/peng/ros_ws/src/qtros/package.xml
Created package file /home/peng/ros_ws/src/qtros/mainpage.dox

Please edit qtros/package.xml and mainpage.dox to finish creating your package
peng@ubuntu:~/ros_ws/src$
```

Through command “tree”, you can take a look at what has been generated for you and their structure. If you don’t have this tool installed, just type in this command to install it,

```
$ sudo apt-get install tree
```

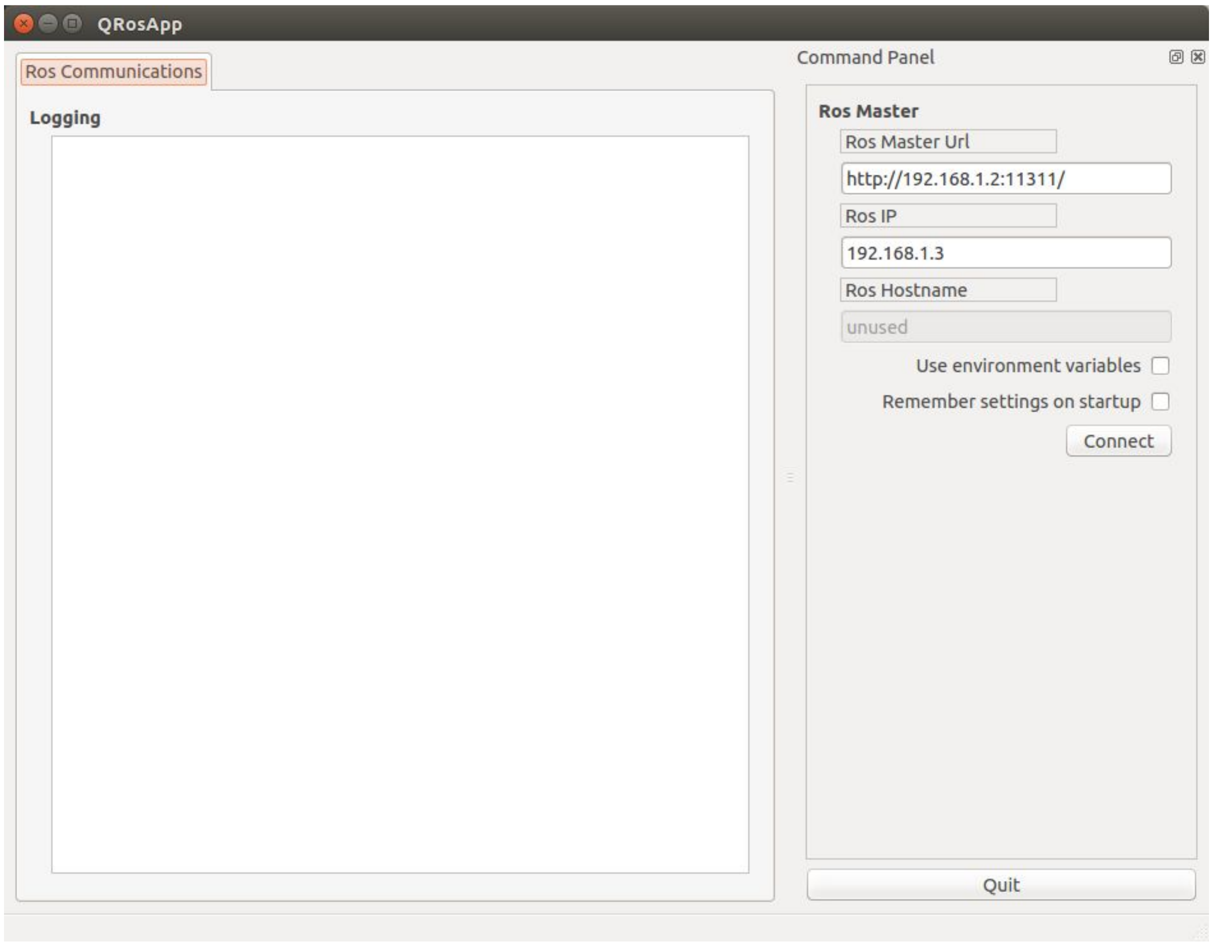


```
peng@ubuntu:~/ros_ws/src$ cd qtros/
peng@ubuntu:~/ros_ws/src/qtros$ tree
.
├── CMakeLists.txt
├── include
│   └── qtros
│       ├── main_window.hpp
│       └── qnode.hpp
├── mainpage.dox
├── package.xml
├── resources
│   ├── images
│   │   └── icon.png
│   └── images.qrc
├── src
│   ├── main.cpp
│   ├── main_window.cpp
│   └── qnode.cpp
└── ui
    └── main_window.ui

6 directories, 11 files
peng@ubuntu:~/ros_ws/src/qtros$
```

Compile and run,

```
$ cd ~/catkin_ws
$ catkin_make
$ rosrun qtros qtros
```

Once running successfully, you can get a GUI window like



Ros Master Url: http://192.168.1.2:11311/
Ros IP: 192.168.1.3
Ros Hostname: unused
Use environment variables ☐
Remember settings on startup ☐
Connect

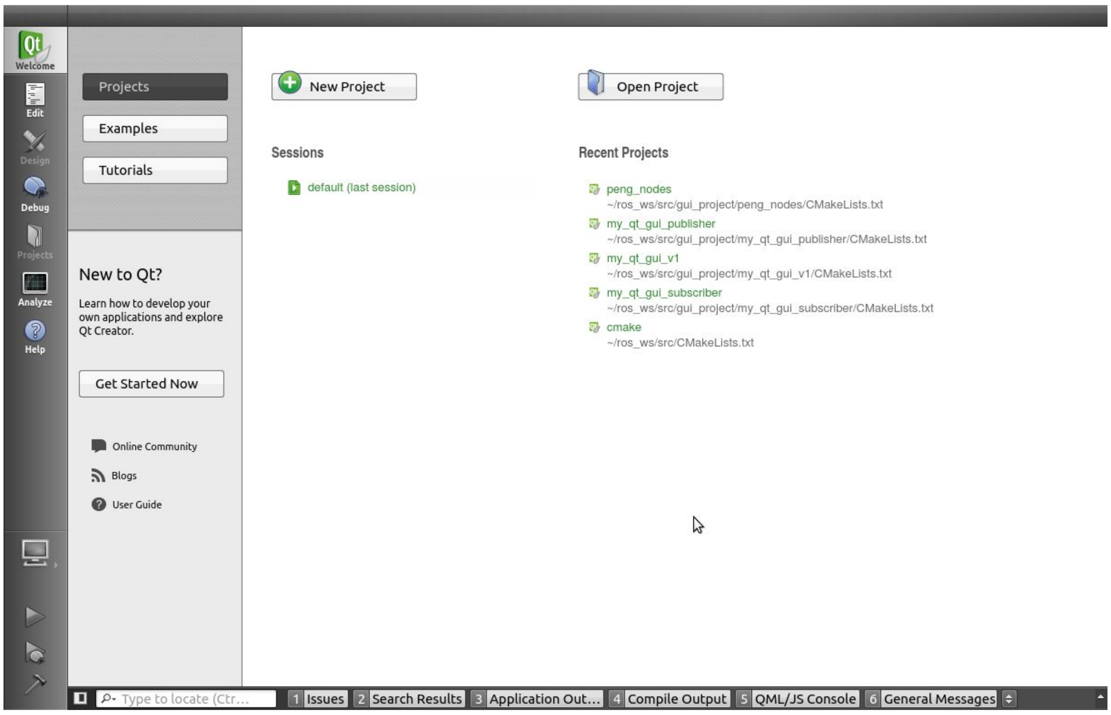
which includes three main features,

- A ROS master chooser
- Separate UI and ROS node classes
- ROS node class has connections for GUI logging and a spinner

Next, you can

- Affect GUI changes in main_window.ui with Qt Designer.
- Modify Qt code in main_window.hpp/main_window.cpp.
- Do ROS topics and services in qnode.hpp/qnode.cpp.

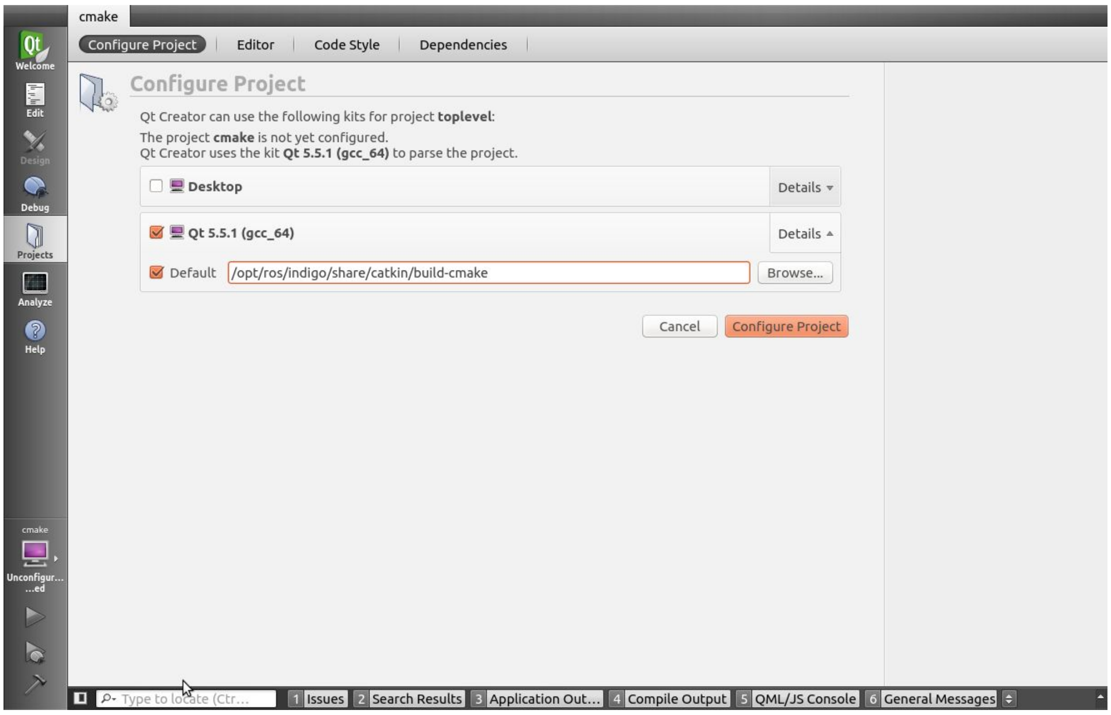
More importantly, we can check the GUI design with Qt Creator. Open the project in Qt Creator by clicking "Open Project" and choosing the CMakeList.txt file in the project folder.



Learn how to develop your own applications and explore Qt Creator.

default (last session)

peng_nodes
~/ros_ws/src/gui_project/peng_nodes/CMakeLists.txt
my_qt_gui_publisher
~/ros_ws/src/gui_project/my_qt_gui_publisher/CMakeLists.txt
my_qt_gui_v1
~/ros_ws/src/gui_project/my_qt_gui_v1/CMakeLists.txt
my_qt_gui_subscriber
~/ros_ws/src/gui_project/my_qt_gui_subscriber/CMakeLists.txt
cmake
~/ros_ws/src/CMakeLists.txt



Configure Project

Qt Creator can use the following kits for project **toplevel**:

The project **cmake** is not yet configured.
Qt Creator uses the kit **Qt 5.5.1 (gcc_64)** to parse the project.

- [ ] **Desktop**
- [x] **Qt 5.5.1 (gcc_64)**
  - [x] Default /opt/ros/indigo/share/catkin/build-cmake

Click “Configure Project”, the project will automatically be configured and compiled. This may generate different a “build” directory.

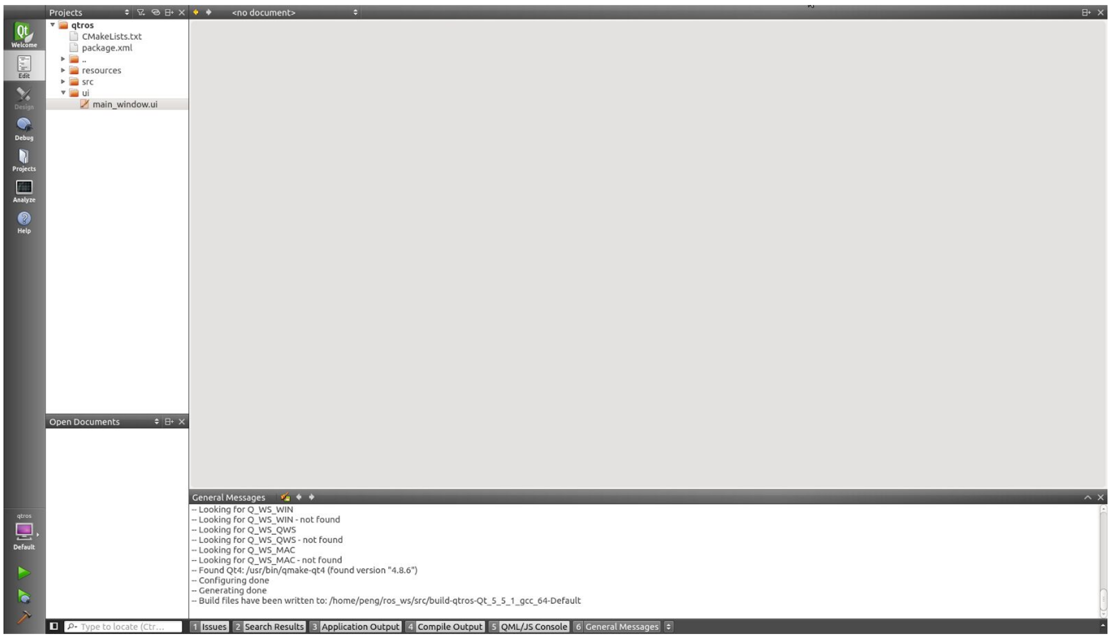
4. Add a Wedget into the GUI

As a strong example, we now try to add new button into the “main_window” and check how it works.

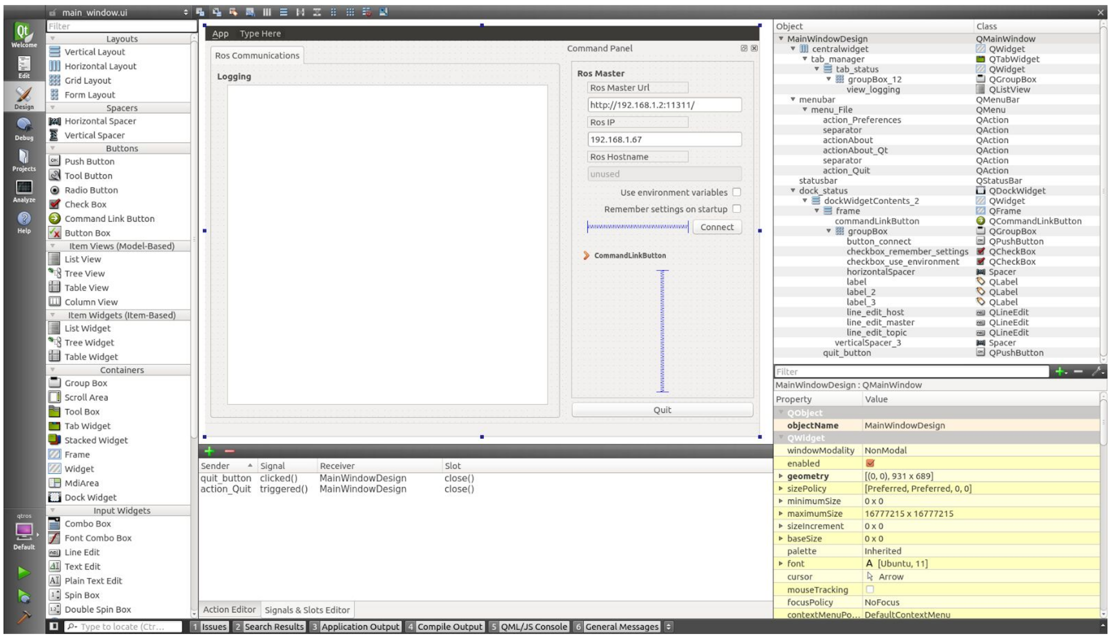
Checking the existing button “Connect” and “Quit”, I was puzzled whether they were automatically linked with the call back functions or they were explicitly triggered. Thus I have these two tests and find both are right answers.

a. button test 1

1) Using “qtros” project created in 3, open “main_window.ui” in UI mode.

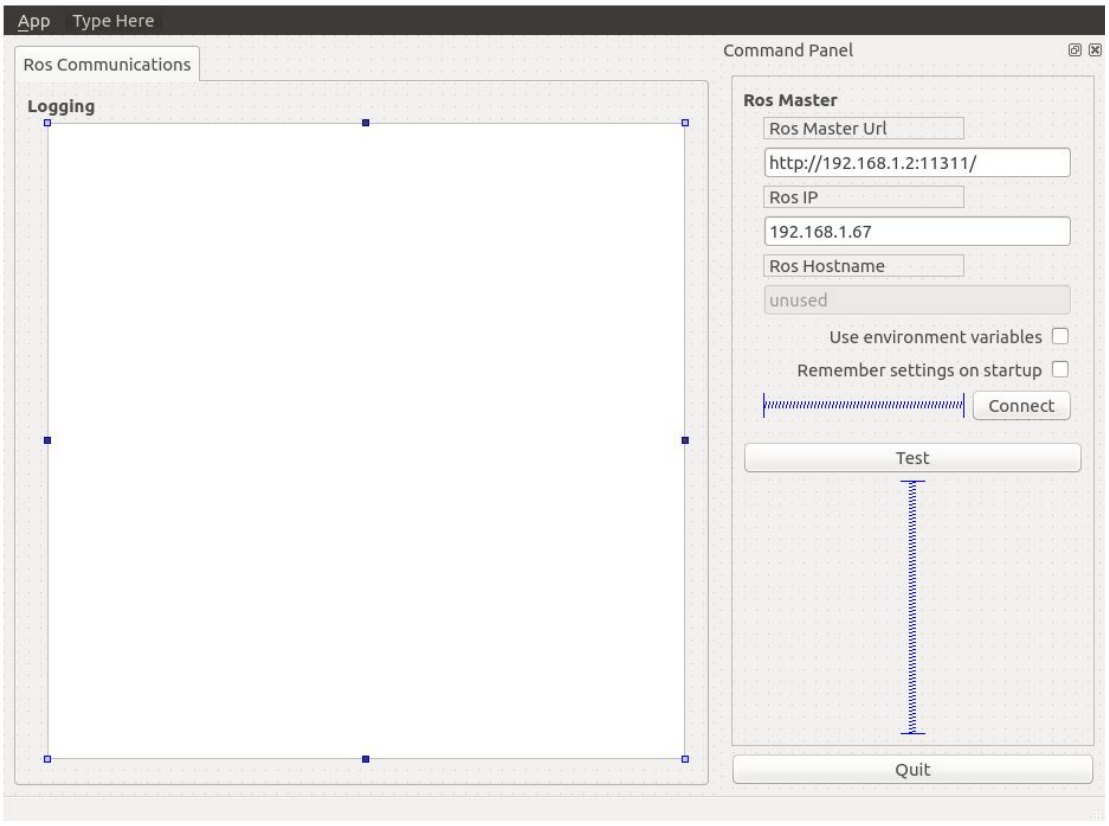


```
-- Looking for Q_WS_WIN
-- Looking for Q_WS_WIN - not found
-- Looking for Q_WS_QWS
-- Looking for Q_WS_QWS - not found
-- Looking for Q_WS_MAC
-- Looking for Q_WS_MAC - not found
-- Found Qt4: /usr/bin/qmake-qt4 (found version "4.8.6")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/peng/ros_ws/src/build-qtros-Qt_5_5_1_gcc_64-Default
```



Sender	Signal	Receiver	Slot
quit_button	clicked()	MainWindowDesign	close()
action_Quit	triggered()	MainWindowDesign	close()

2) Drag a "Push Button" into the ui and replace its text name and object name as "Test" and "button_test", which is in the same way with "button_connect".



3) Open the file "main_window.hpp" and "main_window.cpp" and create two new functions associated with this button test by imitating from the "button_connect" working.

In main_window.hpp

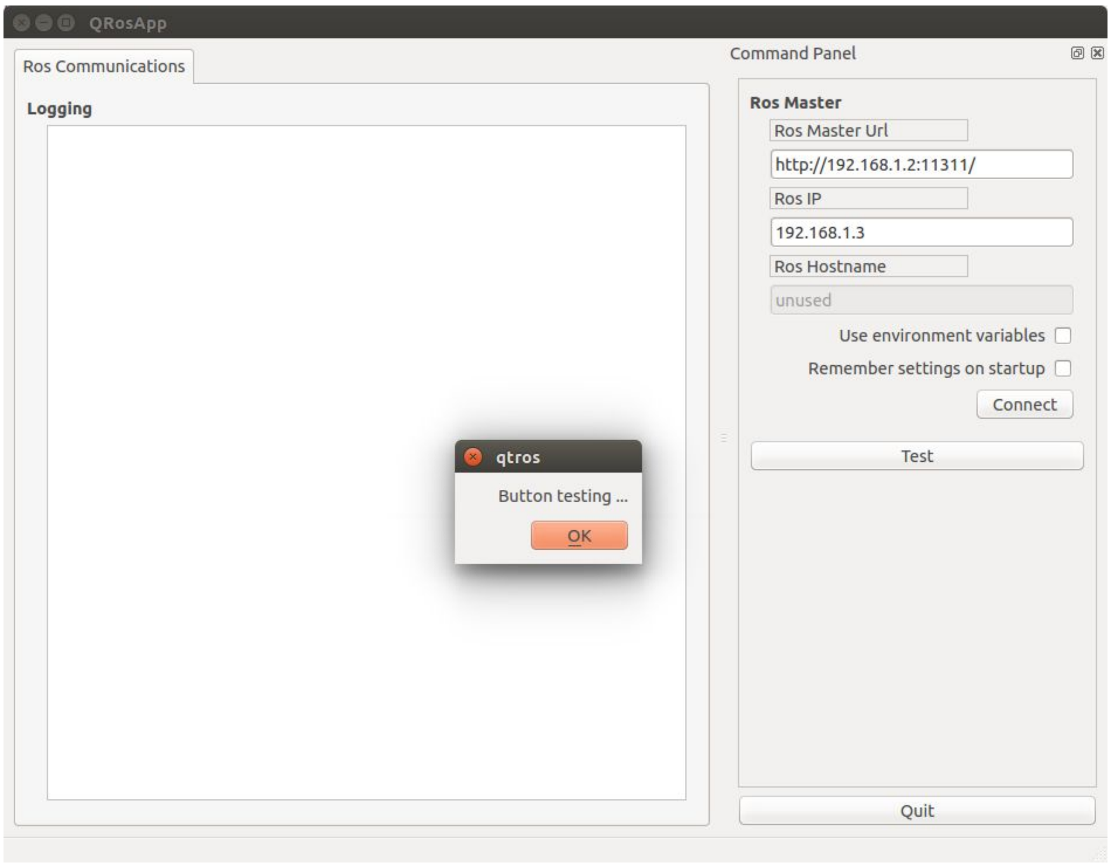
```
public:
…
 void showButtonTestMessage();

public Q_SLOTS:
	/******************************************
	** Auto-connections (connectSlotsByName())
	*******************************************/
…
	void on_button_test_clicked(bool check);
```

In main_window.cpp:

```
/****************************************************************
************
** Implementation [Slots]
*****************************************************************
***********/
…

void MainWindow::showButtonTestMessage() {
    QMessageBox msgBox;
    msgBox.setText("Button test ...");
    msgBox.exec();
    close();
}
…
void MainWindow::on_button_test_clicked(bool check ) {
    showTestButtonMessage();
}
…
```

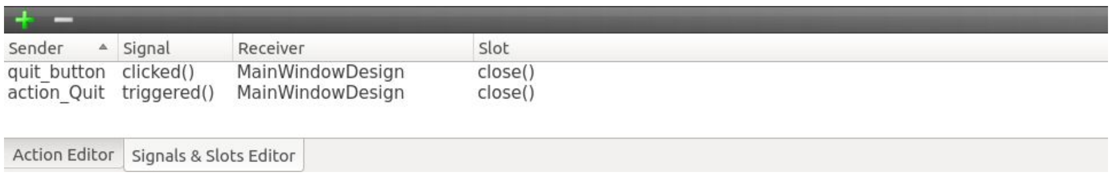
4) Compile and run

We see a new button named “Test”. Then we click it and a message box shows up, which proves the button “Test” is automatically connected by name.



b. button test 2

We notice that the “Quit” button is explicitly connected with a callback function “close()” in Signals & Slots Editor in ui mode.



Sender	Signal	Receiver	Slot
quit_button	clicked()	MainWindowDesign	close()
action_Quit	triggered()	MainWindowDesign	close()

Also, in “main_window.cpp”, there exists some lines seeming to link the widgets and the callback functions together, like

```
    QObject::connect(ui.actionAbout_Qt, SIGNAL(triggered(bool)), qApp,
SLOT(aboutQt())); // qApp is a global variable for the application

QObject::connect(&qnode, SIGNAL(rosShutdown()), this, SLOT(close()));

    QObject::connect(&qnode, SIGNAL(loggingUpdated()), this,
SLOT(updateLoggingView()));
```

So I create a new button, “Left”, to print something different in the logging window. And create a callback function to be called by this button.

In main_window.cpp

```
private:

…

QStringListModel* logging_model;
```

In main_window.cpp

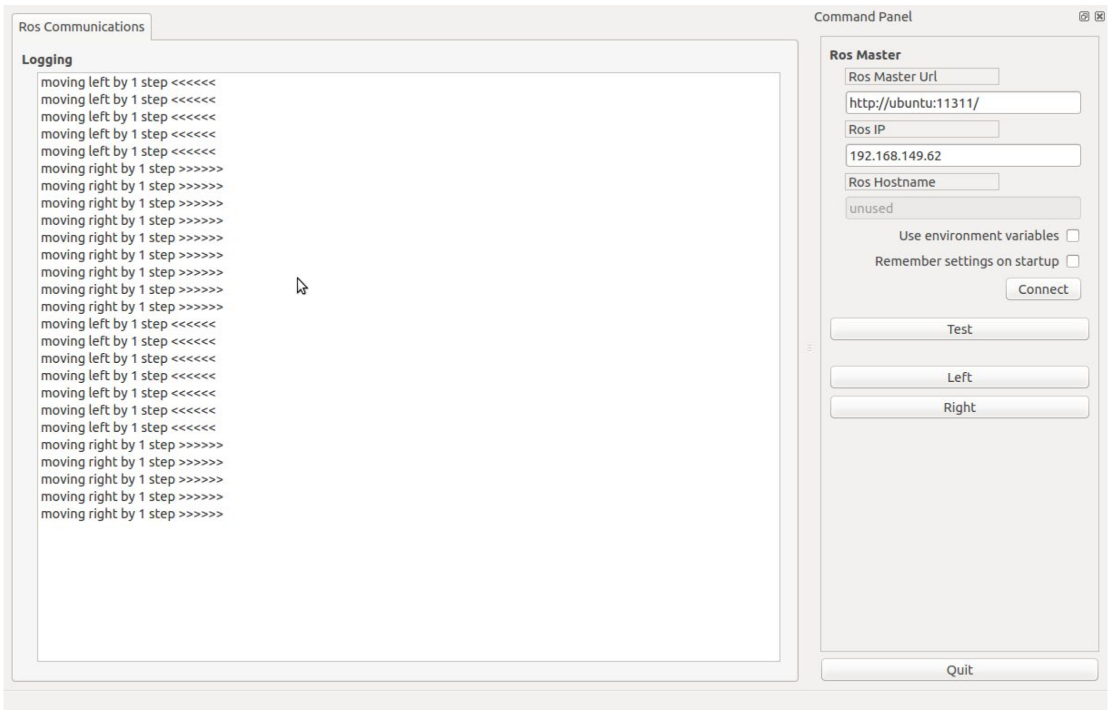
```
MainWindow::MainWindow(int argc, char** argv, QWidget *parent)
      : QMainWindow(parent)
      , qnode(argc,argv)
{
…

    /*******************************
    ** Button test - explicit way
    *******************************/
    QObject::connect(ui.button_left, SIGNAL(clicked()), this,
SLOT(moveLeft()));
}
…

void MainWindow::moveLeft() {
…
}
```

1) Compile and run

   Once the button “left” is clicked, it will show something like “moving left by 1 step <<<<<<”.

   This test proves an explicit way to link callback function to according widget. The key is the line

```
QObject::connect(&qnode, SIGNAL(signal()), this, SLOT(slot()));
```



```
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving left by 1 step <<<<<<
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
moving right by 1 step >>>>>>
```

Ros Master Url: http://ubuntu:11311/
Ros IP: 192.168.149.62
Ros Hostname: unused
Use environment variables ☐
Remember settings on startup ☐
Connect

So basically Qt is using **a signal and slot mechanism**, which is a central feature of Qt and probably the part that differs most from the features provided by other frameworks. You can refer to

http://doc.qt.io/qt-4.8/signalsandslots.html

for more detail.

5. A Publisher and Subscriber example

Populate the qnode.cpp with ROS topics and we can easily build a number of Qt GUI applications in ROS. Here is an example.

By filling in QNode::rosrun() with publisher and subscriber, we can use two nodes to communicate with each other and show everything in the logging windows.

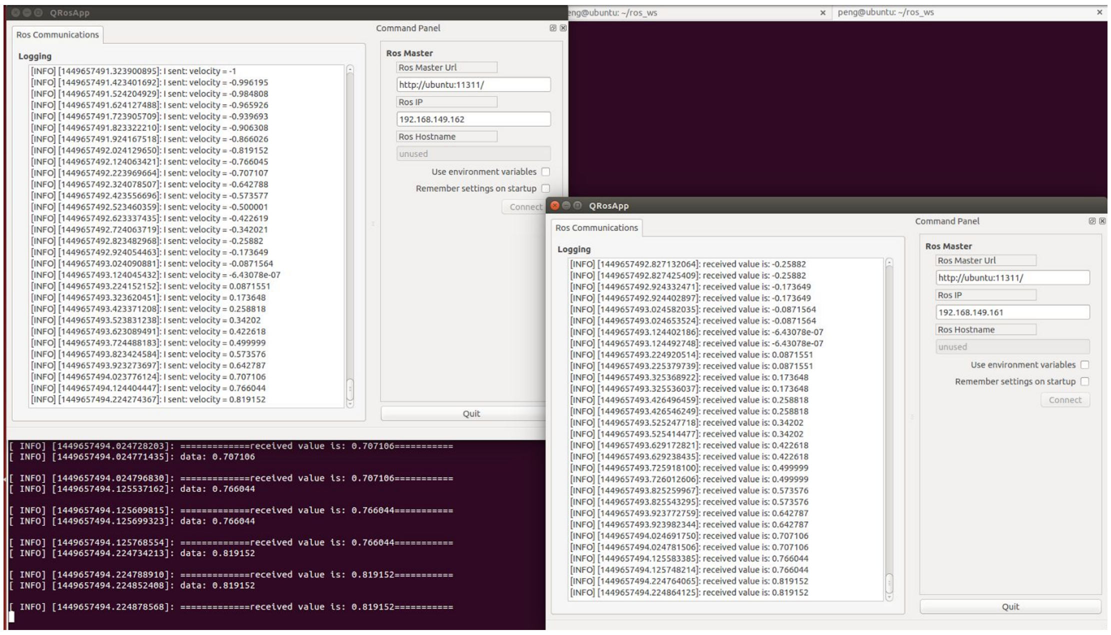
Create two separate packages named “my_gui_publisher” and “my_gui_subsriber”.

In my_gui_publisher/src/qnode.cpp

```
…
chatter_publisher = n.advertise<std_msgs::Float64>("chatter", 1000);
…
```

In my_gui_subscriber/src/qnode.cpp

```
…
chatter_subscriber = n.subscribe("chatter", 1000, &QNode::myCallback,
this);
…
```



```
[INFO] [1449657491.323900895]: I sent: velocity = -1
[INFO] [1449657491.423401692]: I sent: velocity = -0.996195
[INFO] [1449657491.524204929]: I sent: velocity = -0.984808
[INFO] [1449657491.624127488]: I sent: velocity = -0.965926
[INFO] [1449657491.723905709]: I sent: velocity = -0.939693
[INFO] [1449657491.823322210]: I sent: velocity = -0.906308
[INFO] [1449657491.924167518]: I sent: velocity = -0.866026
[INFO] [1449657492.024129650]: I sent: velocity = -0.819152
[INFO] [1449657492.124063421]: I sent: velocity = -0.766045
[INFO] [1449657492.223969664]: I sent: velocity = -0.707107
[INFO] [1449657492.324078507]: I sent: velocity = -0.642788
[INFO] [1449657492.423556696]: I sent: velocity = -0.573577
[INFO] [1449657492.523460359]: I sent: velocity = -0.500001
[INFO] [1449657492.623337435]: I sent: velocity = -0.422619
[INFO] [1449657492.724063719]: I sent: velocity = -0.342021
[INFO] [1449657492.823482968]: I sent: velocity = -0.25882
[INFO] [1449657492.924054463]: I sent: velocity = -0.173649
[INFO] [1449657493.024090881]: I sent: velocity = -0.0871564
[INFO] [1449657493.124045432]: I sent: velocity = -6.43078e-07
[INFO] [1449657493.224152152]: I sent: velocity = 0.0871551
[INFO] [1449657493.323620451]: I sent: velocity = 0.173648
[INFO] [1449657493.423371208]: I sent: velocity = 0.258818
[INFO] [1449657493.523831238]: I sent: velocity = 0.34202
[INFO] [1449657493.623089491]: I sent: velocity = 0.422618
[INFO] [1449657493.724488183]: I sent: velocity = 0.499999
[INFO] [1449657493.823424584]: I sent: velocity = 0.573576
[INFO] [1449657493.923273697]: I sent: velocity = 0.642787
[INFO] [1449657494.023776124]: I sent: velocity = 0.707106
[INFO] [1449657494.124404447]: I sent: velocity = 0.766044
[INFO] [1449657494.224274367]: I sent: velocity = 0.819152
```

Ros Master Url: http://ubuntu:11311/
Ros IP: 192.168.149.162
Ros Hostname: unused
Use environment variables [ ]
Remember settings on startup [ ]

```
[ INFO] [1449657494.024728203]: ============received value is: 0.707106==========
[ INFO] [1449657494.024771435]: data: 0.707106

[ INFO] [1449657494.024796830]: ============received value is: 0.707106==========
[ INFO] [1449657494.125537162]: data: 0.766044

[ INFO] [1449657494.125609815]: ============received value is: 0.766044==========
[ INFO] [1449657494.125699323]: data: 0.766044

[ INFO] [1449657494.125768554]: ============received value is: 0.766044==========
[ INFO] [1449657494.224734213]: data: 0.819152

[ INFO] [1449657494.224788910]: ============received value is: 0.819152==========
[ INFO] [1449657494.224852408]: data: 0.819152

[ INFO] [1449657494.224878568]: ============received value is: 0.819152==========
```

```
[INFO] [1449657492.827132064]: received value is: -0.25882
[INFO] [1449657492.827425409]: received value is: -0.25882
[INFO] [1449657492.924332471]: received value is: -0.173649
[INFO] [1449657492.924402897]: received value is: -0.173649
[INFO] [1449657493.024582035]: received value is: -0.0871564
[INFO] [1449657493.024653524]: received value is: -0.0871564
[INFO] [1449657493.124402186]: received value is: -6.43078e-07
[INFO] [1449657493.124492748]: received value is: -6.43078e-07
[INFO] [1449657493.224920514]: received value is: 0.0871551
[INFO] [1449657493.225379739]: received value is: 0.0871551
[INFO] [1449657493.325368922]: received value is: 0.173648
[INFO] [1449657493.325536037]: received value is: 0.173648
[INFO] [1449657493.426496459]: received value is: 0.258818
[INFO] [1449657493.426546249]: received value is: 0.258818
[INFO] [1449657493.525247718]: received value is: 0.34202
[INFO] [1449657493.525414477]: received value is: 0.34202
[INFO] [1449657493.629172821]: received value is: 0.422618
[INFO] [1449657493.629238435]: received value is: 0.422618
[INFO] [1449657493.725918100]: received value is: 0.499999
[INFO] [1449657493.726012606]: received value is: 0.499999
[INFO] [1449657493.825259967]: received value is: 0.573576
[INFO] [1449657493.825543295]: received value is: 0.573576
[INFO] [1449657493.923772759]: received value is: 0.642787
[INFO] [1449657493.923982344]: received value is: 0.642787
[INFO] [1449657494.024691750]: received value is: 0.707106
[INFO] [1449657494.024781506]: received value is: 0.707106
[INFO] [1449657494.125583385]: received value is: 0.766044
[INFO] [1449657494.125748214]: received value is: 0.766044
[INFO] [1449657494.224764065]: received value is: 0.819152
[INFO] [1449657494.224864125]: received value is: 0.819152
```

Ros Master Url: http://ubuntu:11311/
Ros IP: 192.168.149.161
Ros Hostname: unused
Use environment variables [ ]
Remember settings on startup [ ]

Note: You can use "$ roscore" to check your local ROS MASTER URI and use "$ ifconfig" to inquire your IP address.

6. Conclusion

This tutorial gives some basic steps to start with writing a ROS GUI in C++. Based on these procedures, we can probably build more interactive GUI programs focusing on modifying the qnode files which has been a pure ROS problem.