Qt Development Environment

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Thursday, April 24, 2014

Table of Contents

About This Guide

This document describes the Qt components shipped with the QNX CAR platform, the host system setup needed to develop Qt apps, and the Qt app packaging process.

Although HTML5 is suitable for writing apps that access web services, the Qt components included with the platform provide many services for supporting high-performance, UI-based automotive apps. Using these components, developers proficient with Qt can create user-friendly apps that access car services and data.

To find out about:	See:
The Qt components included in the platform and the capabilities of these components	*Qt Libraries* (p. 9)
How to install and configure the necessary Qt development tools on your host system	*Preparing your host system for Qt development* (p. 15)
How to create Qt apps on your host system and run them on your target system	*Creating and running Qt apps on QNX CAR systems* (p. 35)
The `QtQnxCar2` library (which provides access to car controls) and where to find API documentation for this library	*QtQnxCar2 Library* (p. 11)
The API of the `QPPS` library, which wraps the PPS interface of platform services with a Qt5 interface	*QPPS API* (p. 75)
The API of the `QPlayer` library, which integrates media apps with the `mm-player` media engine	*QPlayer API* (p. 115)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL )`
Command options	`-lR`
Commands	`make`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl** –**Alt** –**Delete**
Keyboard input	`Username`
Keyboard keys	`Enter`
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** **Show View** .

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we use a forward slash (/) as a delimiter in all pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
QNX Qt Development Libraries

The QNX CAR platform ships with the Qt framework and several QNX Qt development libraries that help you write your own Qt apps or Qt-based HMI. If you use a Qt-based HMI, you can run Qt apps and apps written with other technologies, such as HTML5. If your HMI is written with another technology, you can still run Qt apps.

The following QNX Qt development libraries are included with the platform:

QtQnxCar2

This library was developed to support the Qt5 HMI by providing a Qt5 API for accessing middleware services such as navigation, vehicle sensors, and voice-command processing. The library consists of C++ classes with these features:

- Enumerations that define the possible settings for controls
- State information, retrievable through methods
- Signals to notify clients of state changes
- Functions for setting properties or performing domain-specific actions

QPPS

This library provides a Qt5 API for reading from and writing to PPS objects, effectively replacing the POSIX system calls required to access and parse those objects. Using this library, developers can use the standard Qt mechanisms of signals and slots to interact with middleware services. The QPPS library is used by `QtQnxCar2` to communicate with PPS but the library can be used directly by Qt apps.

QPlayer

This special-purpose library integrates the HMI Media Player and other media apps with the platform's media browsing and playback engine, `mm-player`. The `QPlayer` library doesn't interface with PPS, but forwards requests and receives media information through the C API of `mm-player`.

Qt framework

The QNX CAR image includes the runtime binaries and libraries of version 5.2 of the Qt framework. No special configuration or setup is needed to make Qt work on the target system; the version that ships with the platform will run without modification.

The tools for compiling and debugging Qt apps (`Qt Creator` and `qmake`) aren't included in the target image. Instead, the QNX CAR 2.1 installer unpackages these tools onto the host (development) system. For more information on these Qt framework tools, see the *Qt Creator Manual* and *qmake Manual* in the online Qt Project documentation.

QtQnxCar2 Library

The `QtQnxCar2` library provides a Qt5 API to access the automotive subsystems of the QNX CAR platform.

The API consists of more than 50 C++ classes that access services including but not limited to:

- App launching
- Audio and video playback
- Navigation
- Bluetooth
- Car settings (e.g., HVAC, sensors)
- On-screen keyboard
- Automated Speech Recognition (ASR)

The `QtQnxCar2` API reference isn't included in this document. For the location of this API documentation, see the QNX CAR 2.1 release notes.

QPPS Library

The QPPS library wraps the interface to the Persistent Publish Subscribe (PPS) service with a Qt5 interface. With this design, developers can avoid parsing and setting attributes in the PPS objects used by the platform's middleware services (e.g., navigation, radio) and instead use the higher-level Qt5 interface to configure settings for those services.

The library lets you interact with the middleware services at the level of objects and attributes instead of POSIX system calls, which entail working with memory buffers and file descriptors.

PPS object representation

To access a PPS object through the QPPS library, you simply create an `Object` instance in the QPPS API, specifying the path of the PPS object (e.g., `/pps/services/bluetooth/control`) and whether you want your app to publish data to this object, subscribe to updates from this object, or both. Then, you can call the API `Object` methods to set or retrieve one or many attributes at a time in the underlying PPS object.

In addition, the interface provided by QPPS has these capabilities:

- Creation of PPS objects
- Support for boolean, numeric, binary data, string, and JSON-encoded attributes
- Automated notification of a PPS object's attribute changes to all subscribers
- Atomic updates of multiple attributes
- Attribute caching, which allows you to query the latest attribute settings without reading the PPS object
- Troubleshooting functions to test if a PPS object is valid and to retrieve the last error related to the object

Directory monitoring

You can create *DirWatcher* objects to monitor directories for PPS object additions and removals. For example, you can track which devices are attached to your QNX CAR system by monitoring the `/pps/qnx/mount/` directory, which the device publishers update when the user attaches or detaches a hardware device (e.g., a USB stick).

Simulator mode

You can run the library in *simulator mode*, which means it reads from and writes to a simulated object instead of PPS objects in a real filesystem. This mode allows you to develop and test HMI apps on a host system where PPS isn't present.

The simulator mode is transparent to `QPPS` clients. Your Qt app can create an *Object* that represents a particular PPS path, register the *Object* with the *Simulator* object, and then set PPS attributes and receive updates on attribute changes by using the same library calls as when interacting with a real PPS object. Furthermore, you can use the *Simulator* object to update and remove PPS attributes, similar to a platform service that communicates system state changes through PPS.

You can also can monitor PPS directories in simulator mode by creating and using *DirWatcher* objects, in the same way you would monitor directories on a system running PPS.

QPlayer Library

The `QPlayer` library provides a Qt5 API for accessing the `mm-player` media browsing and playback engine. Qt apps can use this library to issue media commands, retrieve browsing results, and read the playback state through `mm-player`.

Unlike most services in the `QtQnxCar2` library, `QPlayer` doesn't use PPS to communicate with the middleware layer because `mm-player` has a C interface. The Media Player in the Qt5 HMI provides a useful reference for integrating a Qt app with the `mm-player` service using `QPlayer`.

The `QPlayer` library abstracts the `mm-player` API into a Qt-compliant object-oriented interface with these features:

- Slots to control playback (e.g., *play()*, *pause()*, *next()*, *previous()*)
- Signals to indicate media events (e.g., a change in playback state or the current track's position)
- A command-based API that allows clients to easily define a main event loop that doesn't block while waiting for slow operations (e.g., browsing on DLNA devices) to complete
- Qt data types to replace C types (e.g., `QString` instead of `char*`)
- C++ classes to replace C structures in the `mm-player` API

Chapter 2
Preparing your host system for Qt development

To write Qt apps, you must install QNX Qt Development Framework (QNX QDF) and Qt Creator 3.0 on your host system and then configure Qt Creator to work with QNX QDF and the target system.

The *host system* is the machine where you develop apps, which can be a Windows or a Linux machine. The *target system* is the machine where you run the apps. In the QNX Qt development environment, the target is a hardware board running the QNX CAR platform.

Before you can configure your host system to support Qt apps, you must have the following:

- An installation of QNX SDP 6.6 on your host system. By default, this platform is installed to `C:\qnx660` on Windows and `/usr/qnx660` on Linux. We refer to this installation location as *DEFAULT_SDP_PATH* throughout this document.
- A target system running QNX CAR 2.1 that's connected to the same network as the host system and has a valid IP address.

Installing QNX QDF

QNX QDF is a collection of Qt header files, libraries, and command-line tools required for building Qt apps. All its content comes from the open-source Qt project and is prebuilt for convenience. You need to install QNX QDF before you can develop Qt apps that target QNX CAR 2.1.

To install the QNX QDF:

1. Download the Qt archive file appropriate for your host system OS by going to our website, *`www.qnx.com`*, logging into your myQNX account, and then going to the Download area.

   To find the archive file containing the Qt development tools that will work with your host, search for files with names similar to "`QT 5.2 (Windows Host Tools)`" or "`QT 5.2 (Linux Host Tools)`" in the download area of the QNX website.

2. Open the archive file and navigate one level down from the root directory to access the directory containing the QNX QDF files.

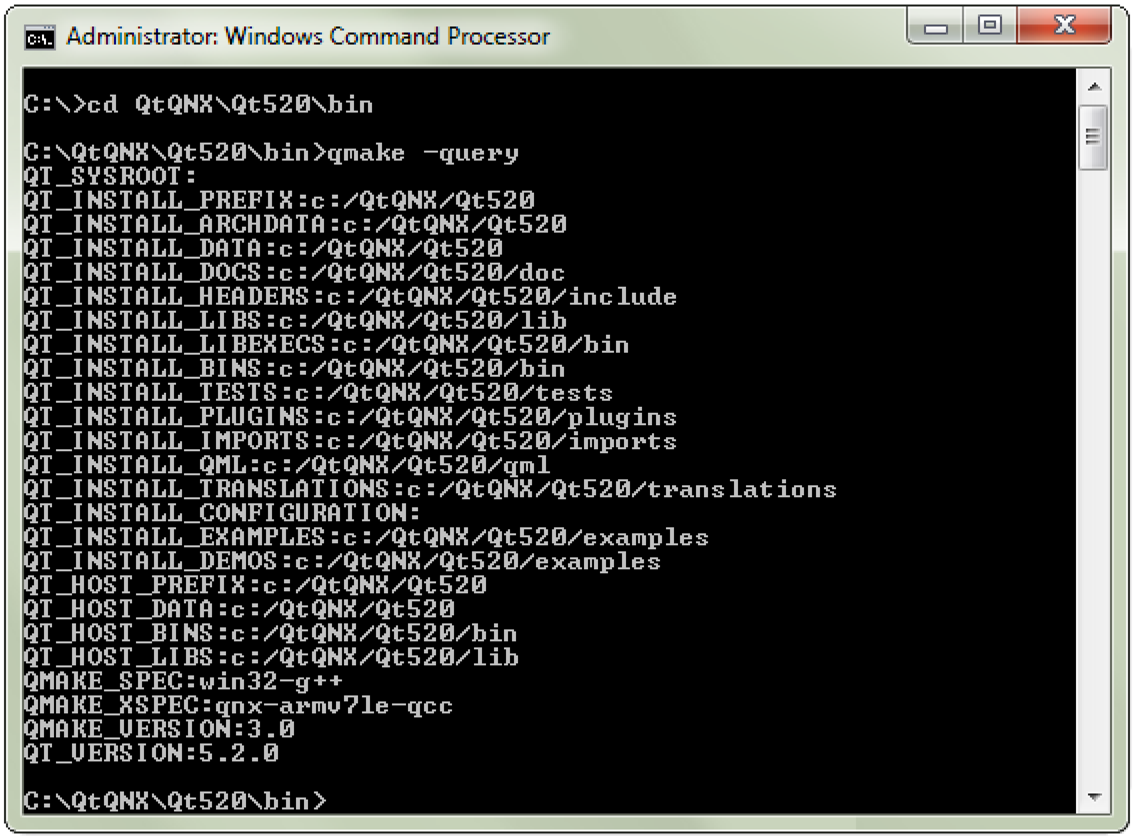
   In the Windows archive, this will be the `QtQNX` directory, found within the top-level `qt-windows-armle-v7` directory. For Linux, this will be the `qt5-5.2` directory, found within `qt-linux-armle-v7`.

3. Unzip this directory to `C:\` on Windows or `/` on Linux.

   QNX QDF must be unzipped to this particular location because some Qt tools have hardcoded paths. Installing the package to another location will result in compilation and build errors when developing a Qt project. For Windows, your development projects must also be on `C:`.

   (Optional)

4. Verify the correctness of the QNX QDF path by opening an OS terminal, navigating to `C:\QtQNX\Qt520\bin` on Windows or `/base/qt5-5.2/bin` on Linux, and typing `qmake -query`:



Administrator: Windows Command Processor

```
C:\>cd QtQNX\Qt520\bin

C:\QtQNX\Qt520\bin>qmake -query
QT_SYSROOT:
QT_INSTALL_PREFIX:c:/QtQNX/Qt520
QT_INSTALL_ARCHDATA:c:/QtQNX/Qt520
QT_INSTALL_DATA:c:/QtQNX/Qt520
QT_INSTALL_DOCS:c:/QtQNX/Qt520/doc
QT_INSTALL_HEADERS:c:/QtQNX/Qt520/include
QT_INSTALL_LIBS:c:/QtQNX/Qt520/lib
QT_INSTALL_LIBEXECS:c:/QtQNX/Qt520/bin
QT_INSTALL_BINS:c:/QtQNX/Qt520/bin
QT_INSTALL_TESTS:c:/QtQNX/Qt520/tests
QT_INSTALL_PLUGINS:c:/QtQNX/Qt520/plugins
QT_INSTALL_IMPORTS:c:/QtQNX/Qt520/imports
QT_INSTALL_QML:c:/QtQNX/Qt520/qml
QT_INSTALL_TRANSLATIONS:c:/QtQNX/Qt520/translations
QT_INSTALL_CONFIGURATION:
QT_INSTALL_EXAMPLES:c:/QtQNX/Qt520/examples
QT_INSTALL_DEMOS:c:/QtQNX/Qt520/examples
QT_HOST_PREFIX:c:/QtQNX/Qt520
QT_HOST_DATA:c:/QtQNX/Qt520
QT_HOST_BINS:c:/QtQNX/Qt520/bin
QT_HOST_LIBS:c:/QtQNX/Qt520/lib
QMAKE_SPEC:win32-g++
QMAKE_XSPEC:qnx-armv7le-qcc
QMAKE_VERSION:3.0
QT_VERSION:5.2.0

C:\QtQNX\Qt520\bin>
```

The installation location should match the first few directory levels in the paths listed in the output.

Installing Qt Creator

Qt Creator is the IDE that you use to develop Qt apps. The IDE includes a code editor, visual debugger, and integrated UI layout and forms designer. QNX CAR 2.1 supports apps written with version 3.0 of Qt Creator.

To install Qt Creator 3.0:

1. Download the archive file appropriate for your host system OS from *http://qt-project.org/downloads#qt-creator* to your host system.

   Although Qt Creator is included in the binary packages for Qt 5.2.0, it's not necessary to install this library.

2. Run the downloaded archive file (which is an `.exe` file on Windows and a `.run` file on Linux) and install the product according to the installer's instructions.

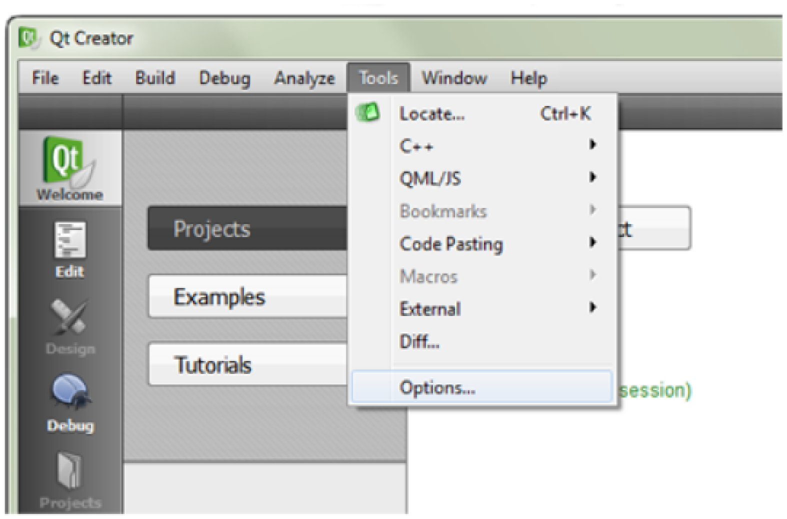
Qt Creator 3.0 is installed. Before you can develop Qt apps, you must configure a QNX device to represent your target system and a toolchain to define your compiler and debugger settings. The sections that follow explain how to do this.

Configuring a QNX device in Qt Creator

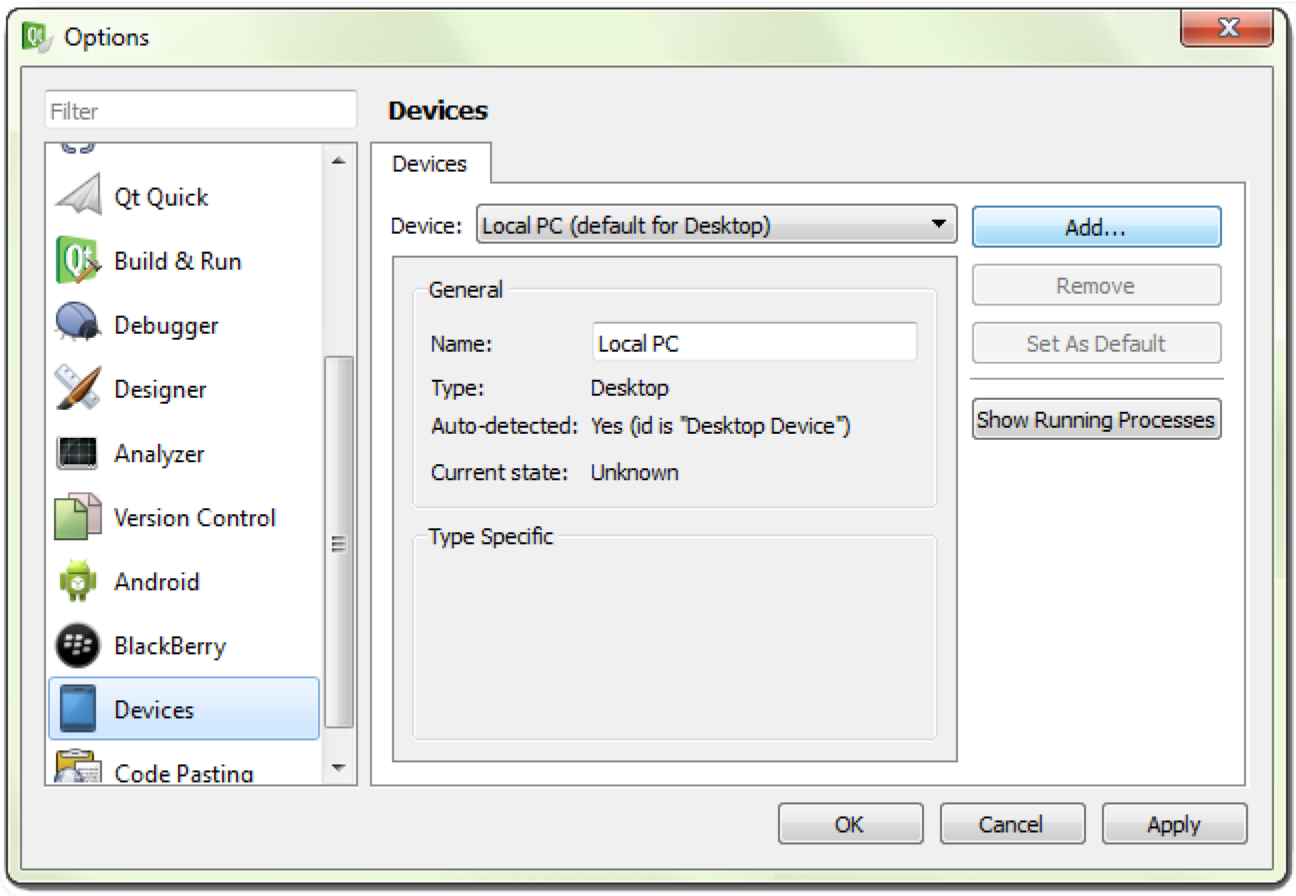
You must configure a QNX device to tell Qt Creator which target system your apps will be deployed onto. In the QNX Qt development environment, the target is your hardware board running QNX CAR 2.1.

To configure a QNX device in Qt Creator:

1. In the IDE, select the **Tools** menu, then click **Options** to open the **Options** dialog.



2. Choose **Devices** in the left-side menu and click the **Add...** button on the right side.



Devices

Device: Local PC (default for Desktop)

General

Name: Local PC

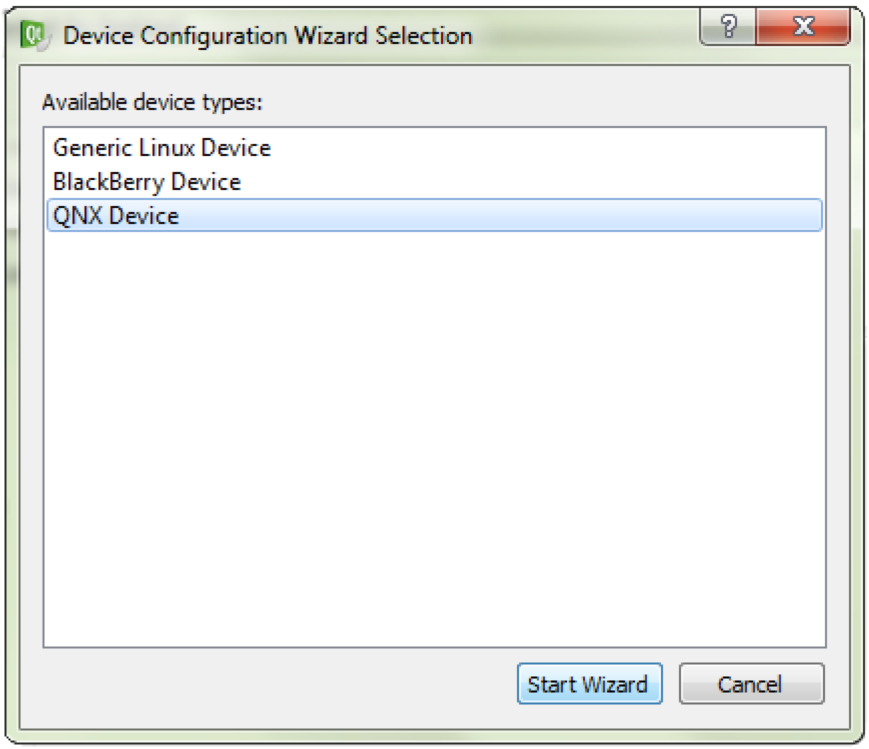
Type: Desktop

Auto-detected: Yes (id is "Desktop Device")

Current state: Unknown

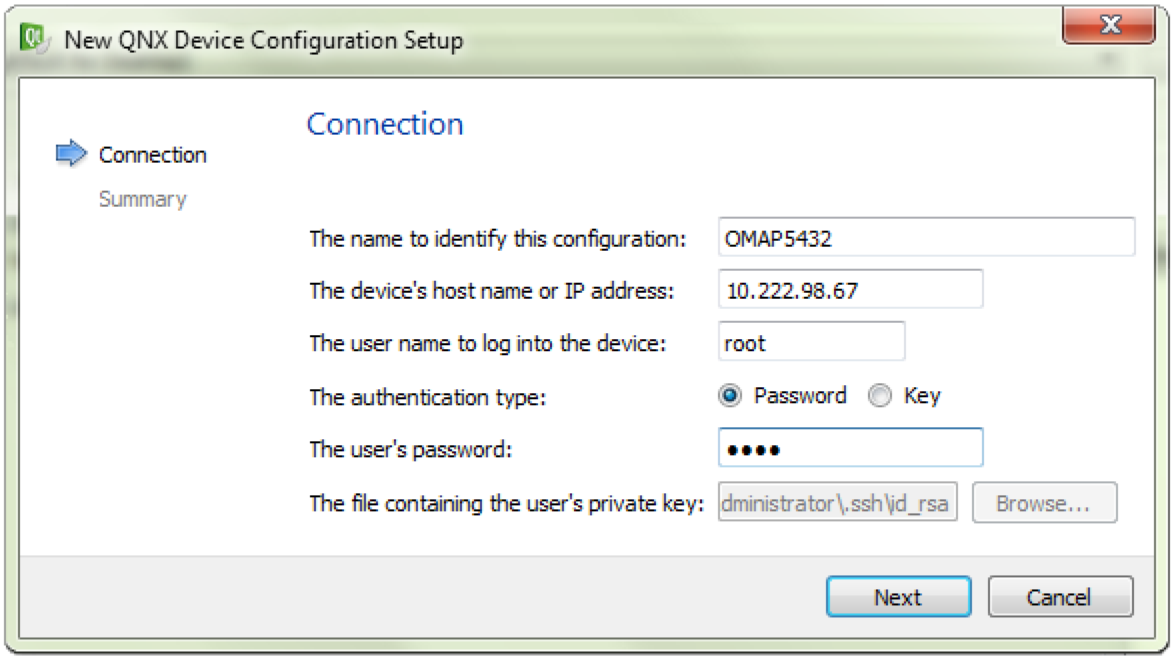
Type Specific

3. In the **Device Configuration Wizard Selection** dialog, choose `QNX Device` and click **Start Wizard**.



4. In the **New QNX Device Configuration Setup** dialog, fill in the connection fields:
   a. Name the device configuration something meaningful, like `OMAP5432`.
   b. Enter the IP address of the target board.
   c. In each of the username and password fields, enter `root`.

      To display this last field, ensure you've selected **Password** as the authentication type.
   d. Click **Next**.



Connection

The name to identify this configuration: OMAP5432

The device's host name or IP address: 10.222.98.67

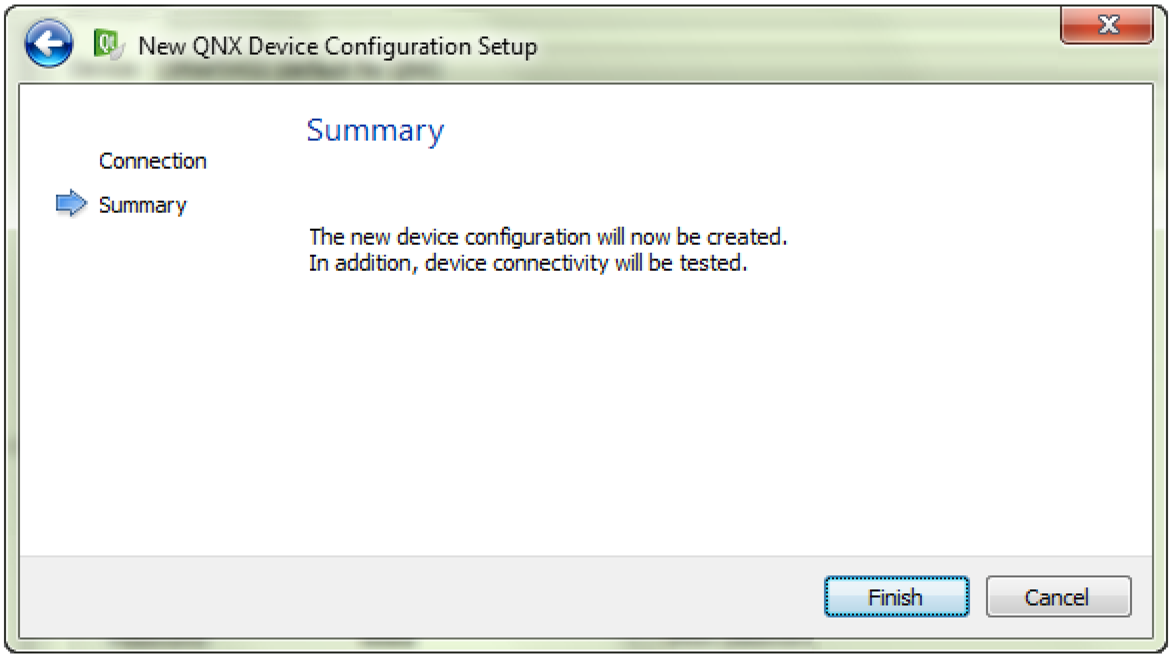
The user name to log into the device: root

The authentication type: [✓ Password] Key

The user's password: ••••

The file containing the user's private key: dministrator\.ssh\id_rsa

5. On the summary page, click **Finish**.

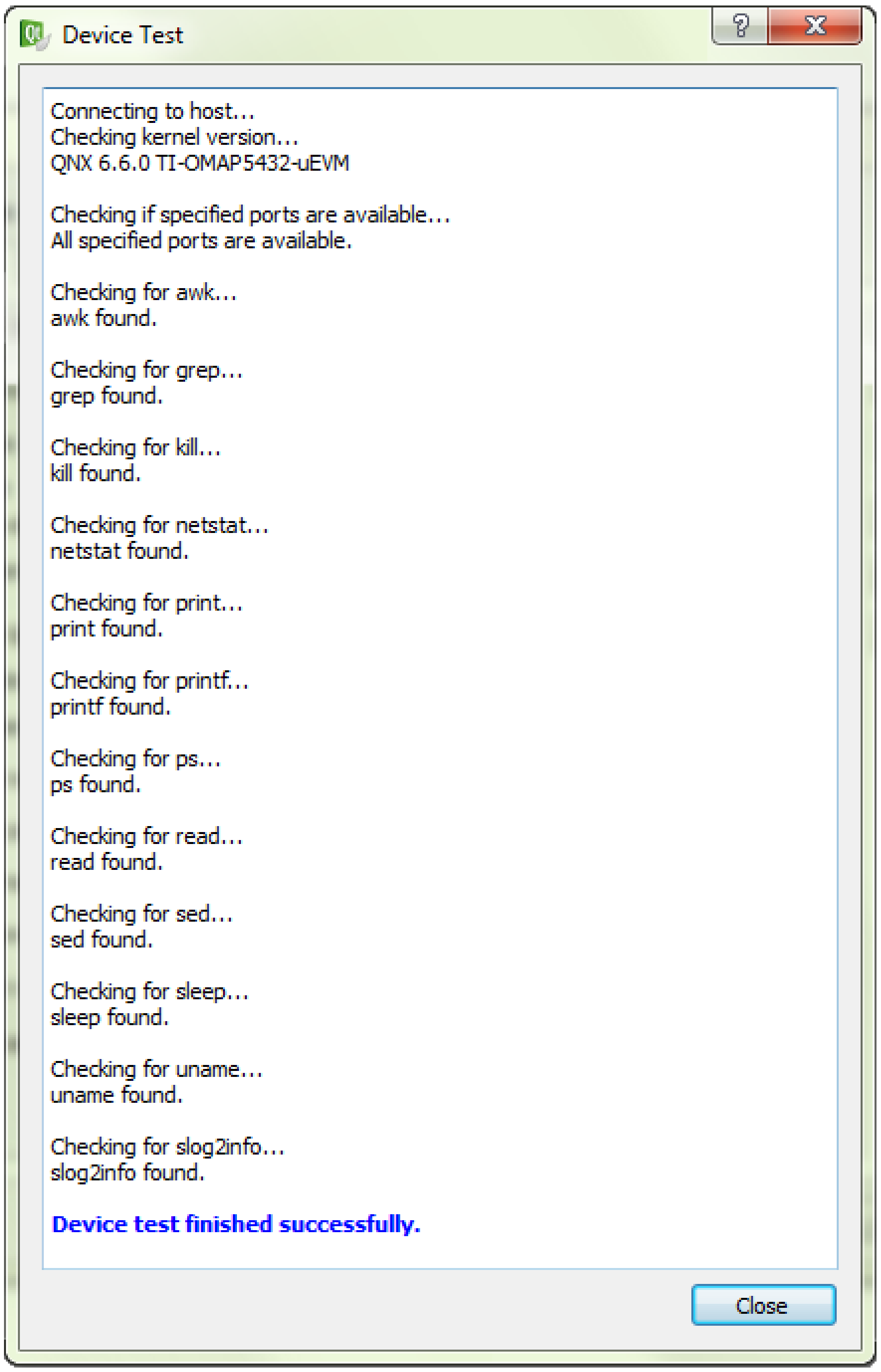


Summary

The new device configuration will now be created.
In addition, device connectivity will be tested.

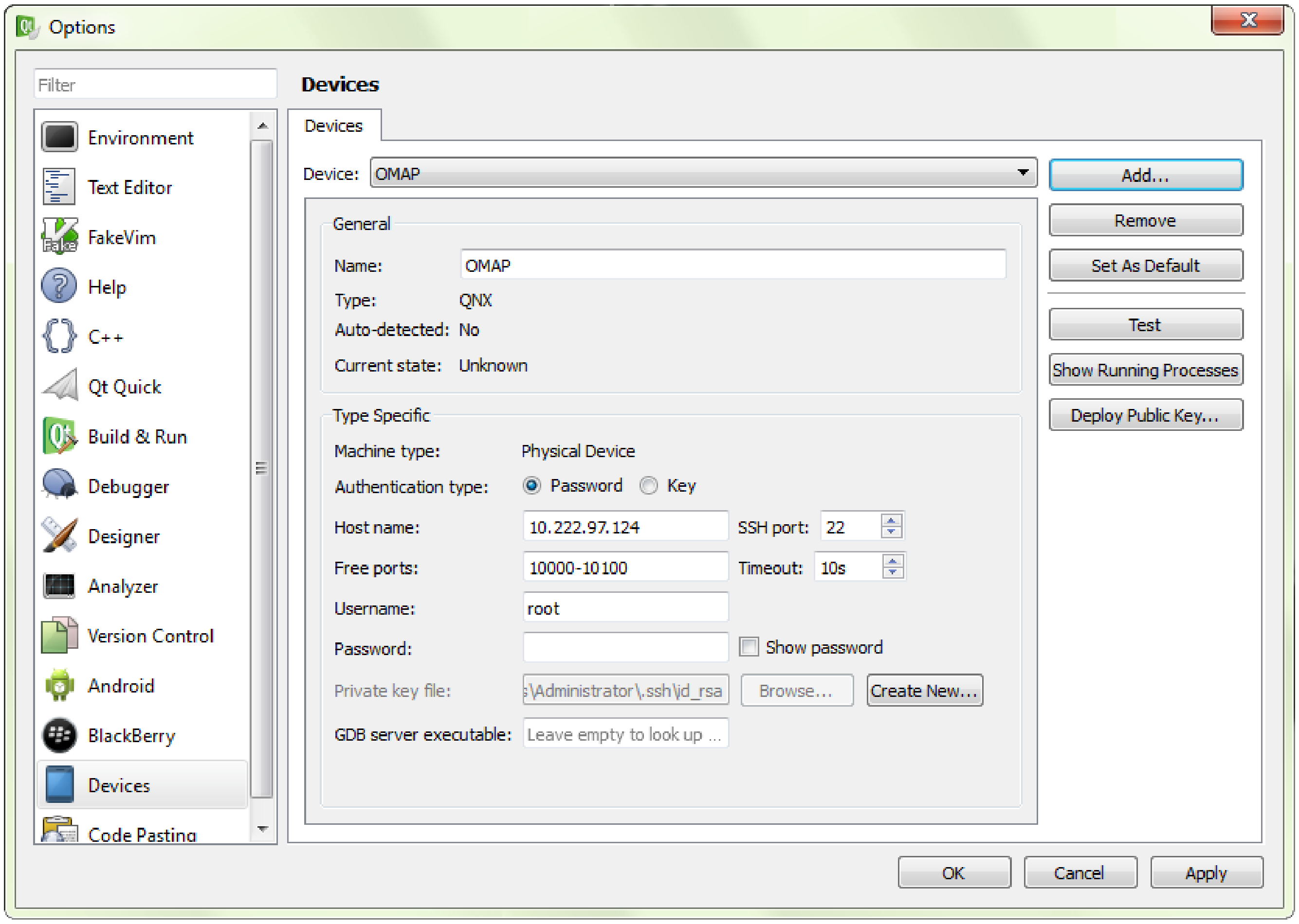
Qt Creator starts the device connectivity test, which entails connecting to the newly configured device and checking if the specified ports and certain key services (e.g., `grep`, `awk`) are available.

6. Examine the test results in the **Device Test** dialog, then click **Close** to return to the **Options** dialog.



```
Connecting to host...
Checking kernel version...
QNX 6.6.0 TI-OMAP5432-uEVM

Checking if specified ports are available...
All specified ports are available.

Checking for awk...
awk found.

Checking for grep...
grep found.

Checking for kill...
kill found.

Checking for netstat...
netstat found.

Checking for print...
print found.

Checking for printf...
printf found.

Checking for ps...
ps found.

Checking for read...
read found.

Checking for sed...
sed found.

Checking for sleep...
sleep found.

Checking for uname...
uname found.

Checking for slog2info...
slog2info found.

Device test finished successfully.
```

7. If the test failed, review the new device's connection settings (now shown in the **Devices** tab) and fix any improper settings. You can then click **Test** (on the right side) to retest your device (this action relaunches the **Device Test** dialog and you would then go back to Step *6* (p. 22)).



Devices

Device: OMAP

General

Name: OMAP
Type: QNX
Auto-detected: No
Current state: Unknown

Type Specific

Machine type: Physical Device
Authentication type: [✓ Password] Key
Host name: 10.222.97.124 SSH port: 22
Free ports: 10000-10100 Timeout: 10s
Username: root
Password: Show password
Private key file: :\Administrator\.ssh\id_rsa
GDB server executable: Leave empty to look up ...

8. Click the **OK** button in the bottom right corner to close the **Options** dialog.

You must close the **Options** dialog and return to the main application screen before relaunching the same dialog and configuring the build and run settings; otherwise, the new device won't be listed. Clicking **Apply** isn't enough to save the new device configuration. This is a known issue in Qt Creator.

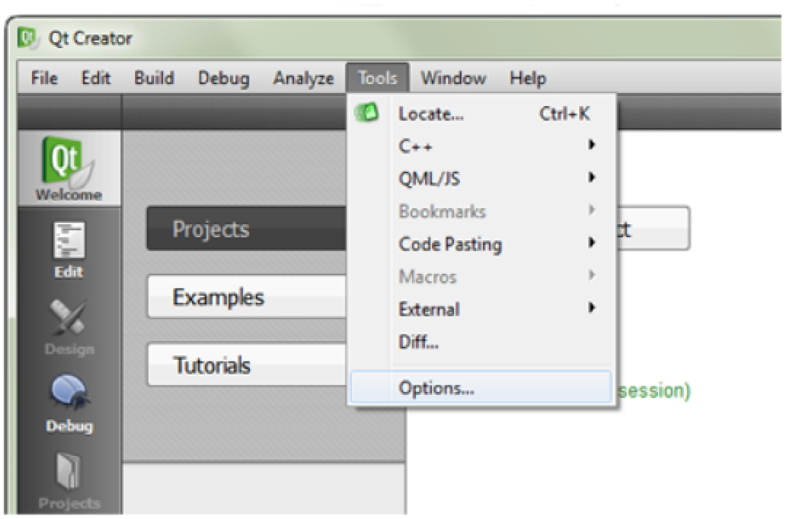
Qt Creator has a device profile representing your target system. You can now configure a toolchain.

Configuring a toolchain in Qt Creator

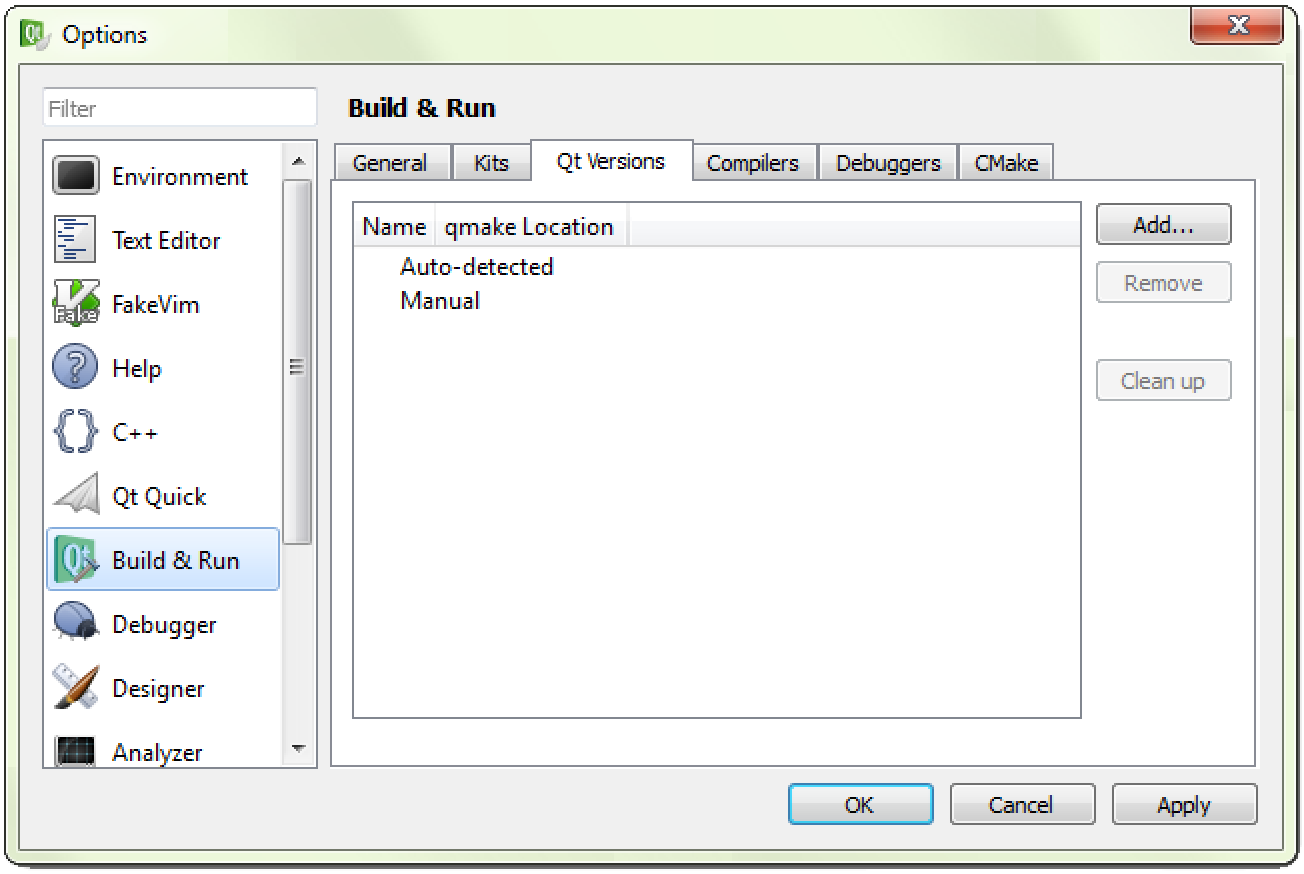
After defining a QNX device to represent your target system, you must set up a toolchain in Qt Creator. The toolchain defines the build and run environment based on the QNX QDF installation and the compiler, debugger, and target device configurations.

To configure a toolchain in Qt Creator:

1. In the IDE, select the **Tools** menu, then click **Options** to open the **Options** dialog.

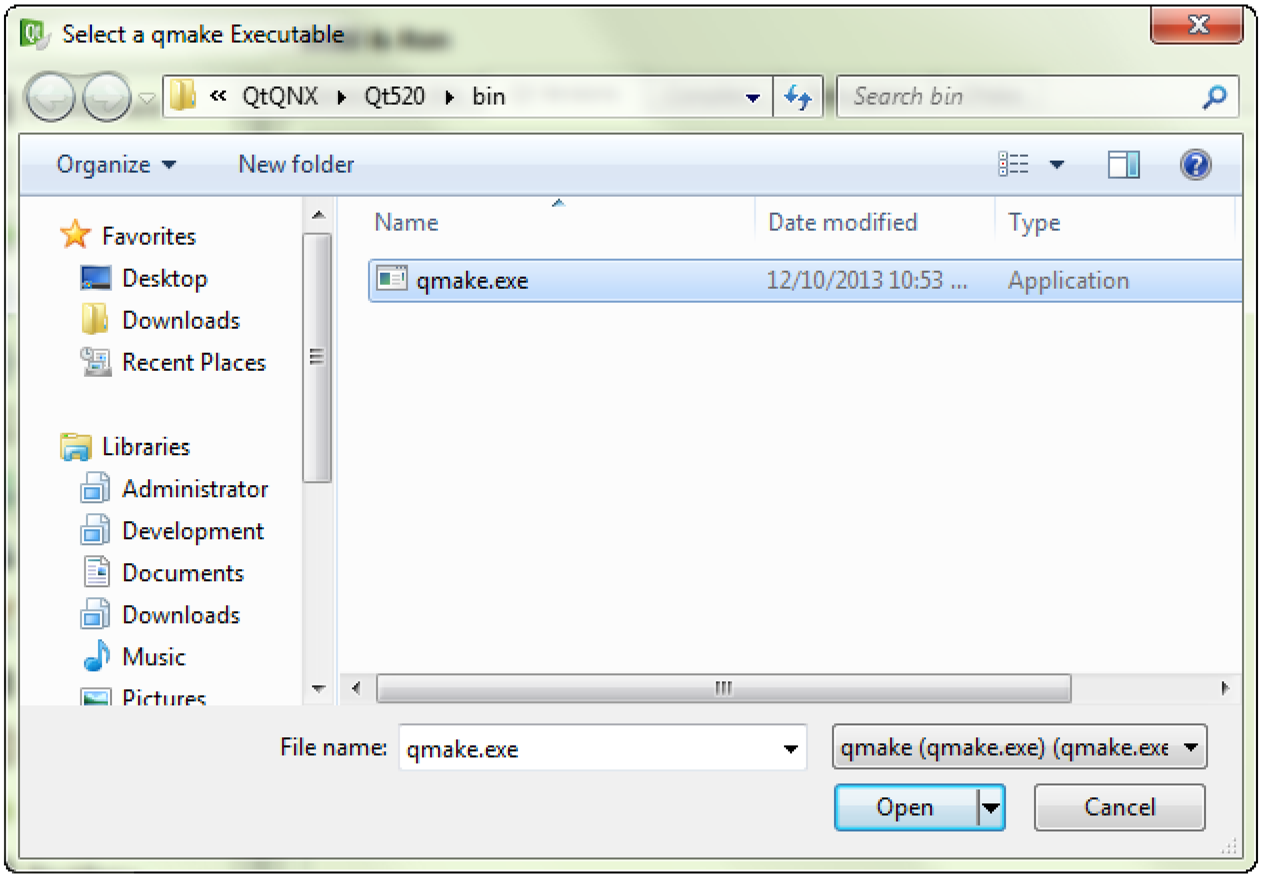


2. Choose **Build & Run** in the left-side menu, click the **Qt Versions** tab in the main viewing area, then click the **Add...** button on the right side.



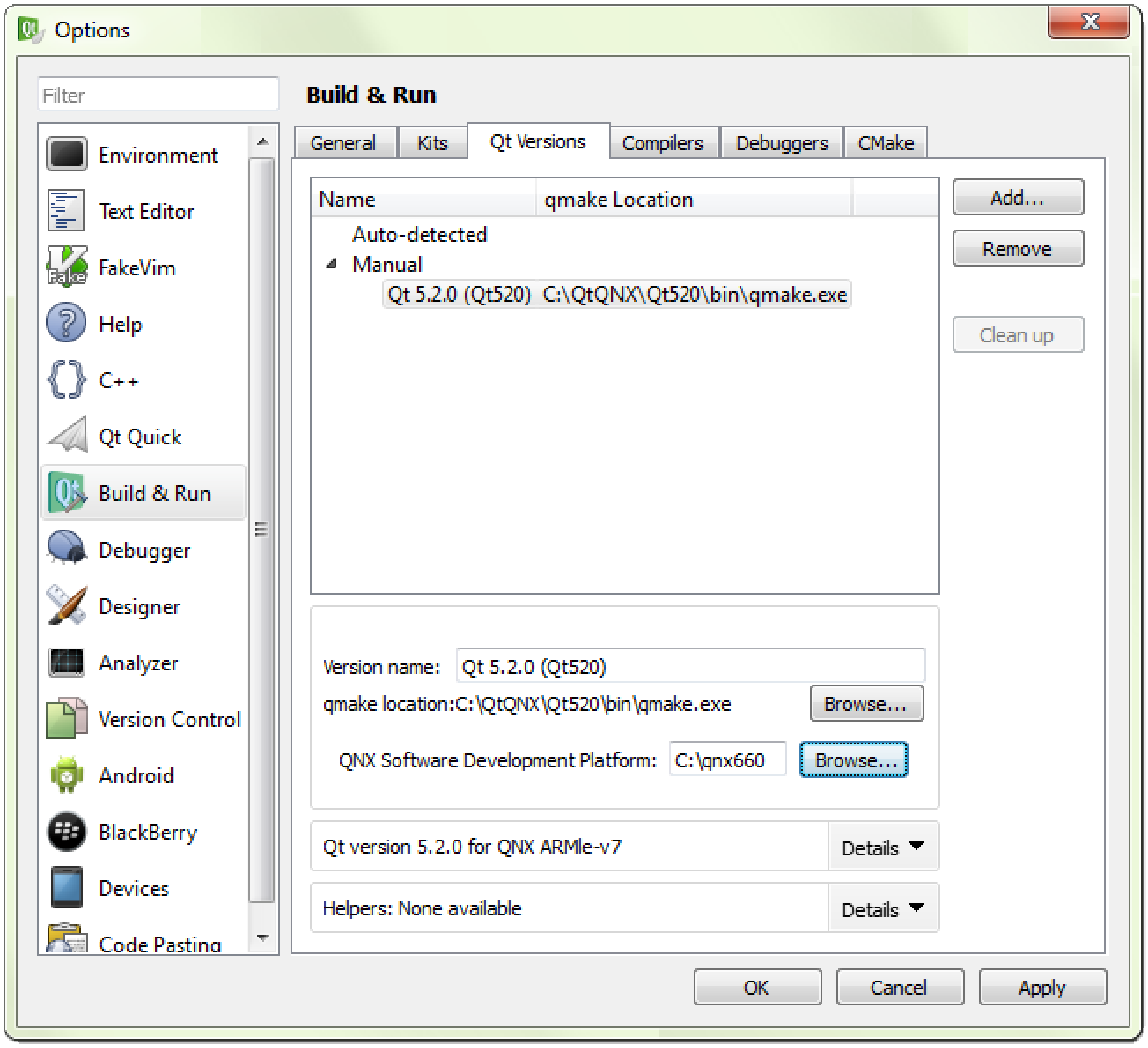
The IDE opens a file selector.

3. In the file selector, either navigate to `C:\QtQNX\Qt520\bin` and select `qmake.exe` (on Windows) or navigate to `/base/qt5-5.2/bin` and select `qmake` (on Linux), then click **Open**.



The **Options** dialog displays additional fields for configuring the selected Qt version.

4. At the bottom of the dialog, on the line that reads `QNX Software Development Platform`, click **Browse...**.



Name	qmake Location
Auto-detected	
Manual	
Qt 5.2.0 (Qt520)	C:\QtQNX\Qt520\bin\qmake.exe

Version name: Qt 5.2.0 (Qt520)

qmake location:C:\QtQNX\Qt520\bin\qmake.exe

QNX Software Development Platform: C:\qnx660

Qt version 5.2.0 for QNX ARMle-v7

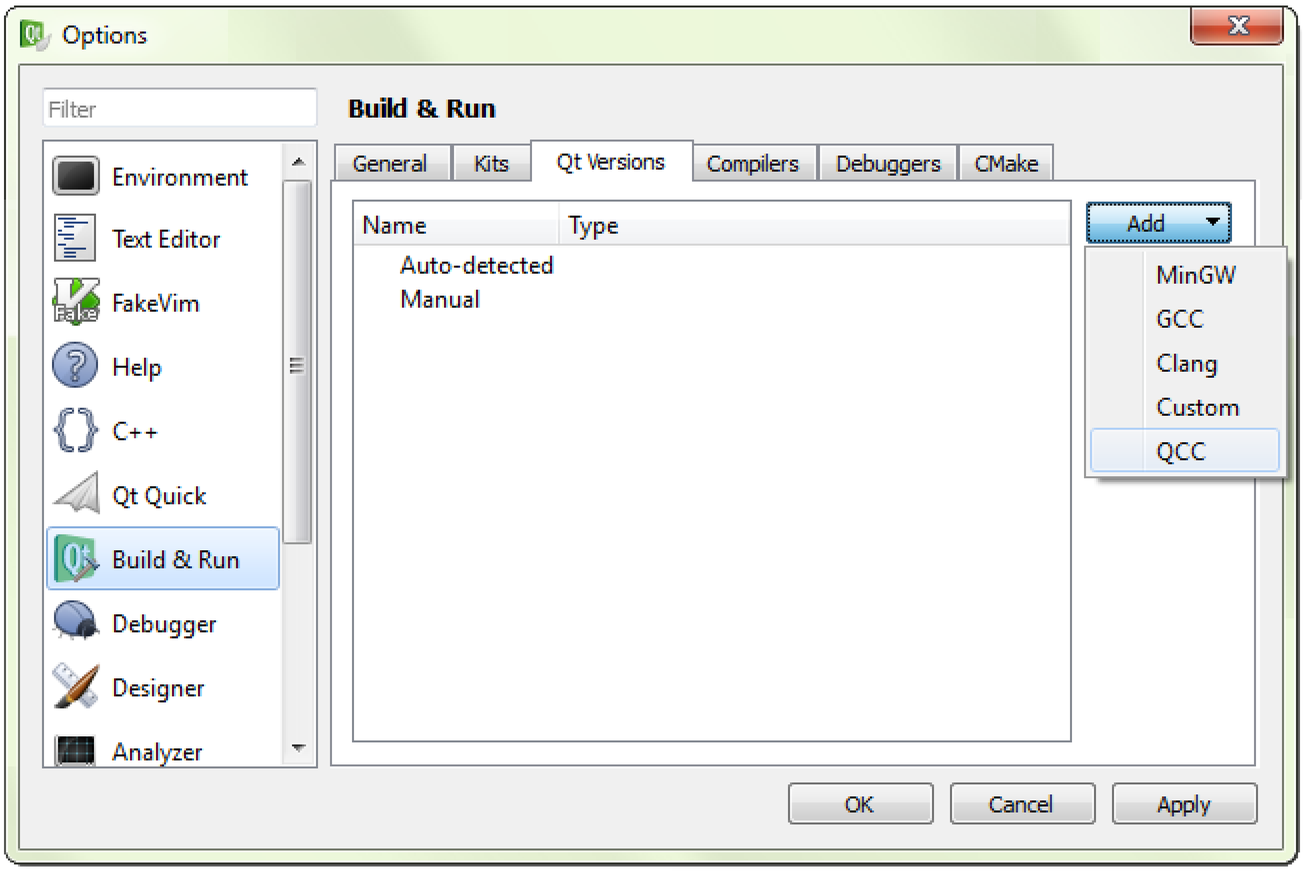
Helpers: None available

The IDE opens another file selector.

5. Navigate to *DEFAULT_SDP_PATH* and click **Select Folder**.

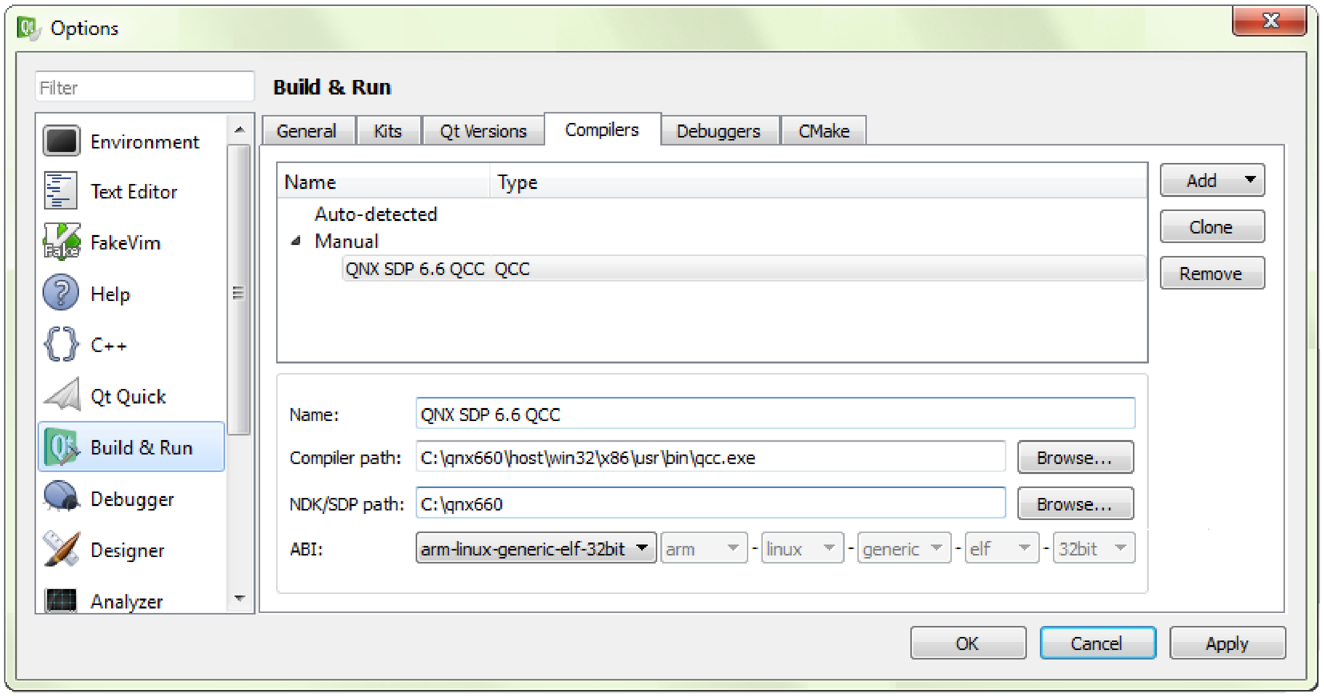
   The `QNX Software Development` field now lists the directory containing the QNX SDP 6.6 installation on your host system.

6. Click the **Compilers** tab, click the **Add** button on the right side, then select `QCC` from the dropdown list.



The **Options** dialog displays additional fields at the bottom for configuring the newly added compiler.

7. Fill in the compiler fields:
   a. In the **Name** field, enter `QNX SDP 6.6 QCC`.
   b. On the `Compiler path` line, click **Browse...** to open the file selector. On Windows, navigate to *DEFAULT_SDP_PATH*`\host\win32\x86\usr\bin` and choose `qcc.exe`. On Linux, navigate to *DEFAULT_SDP_PATH*`/host/linux/x86/usr/bin` and choose `qcc`. Click **Select Folder** to confirm the setting.
   c. On the `NDK/SDP path` line, click **Browse...** to open the file selector, navigate to *DEFAULT_SDP_PATH*, then click **Select Folder**.
   d. In the dropdown list for **ABI**, select `arm-linux-generic-elf-32bit`.



Name	Type
Auto-detected	
Manual	
QNX SDP 6.6 QCC	QCC

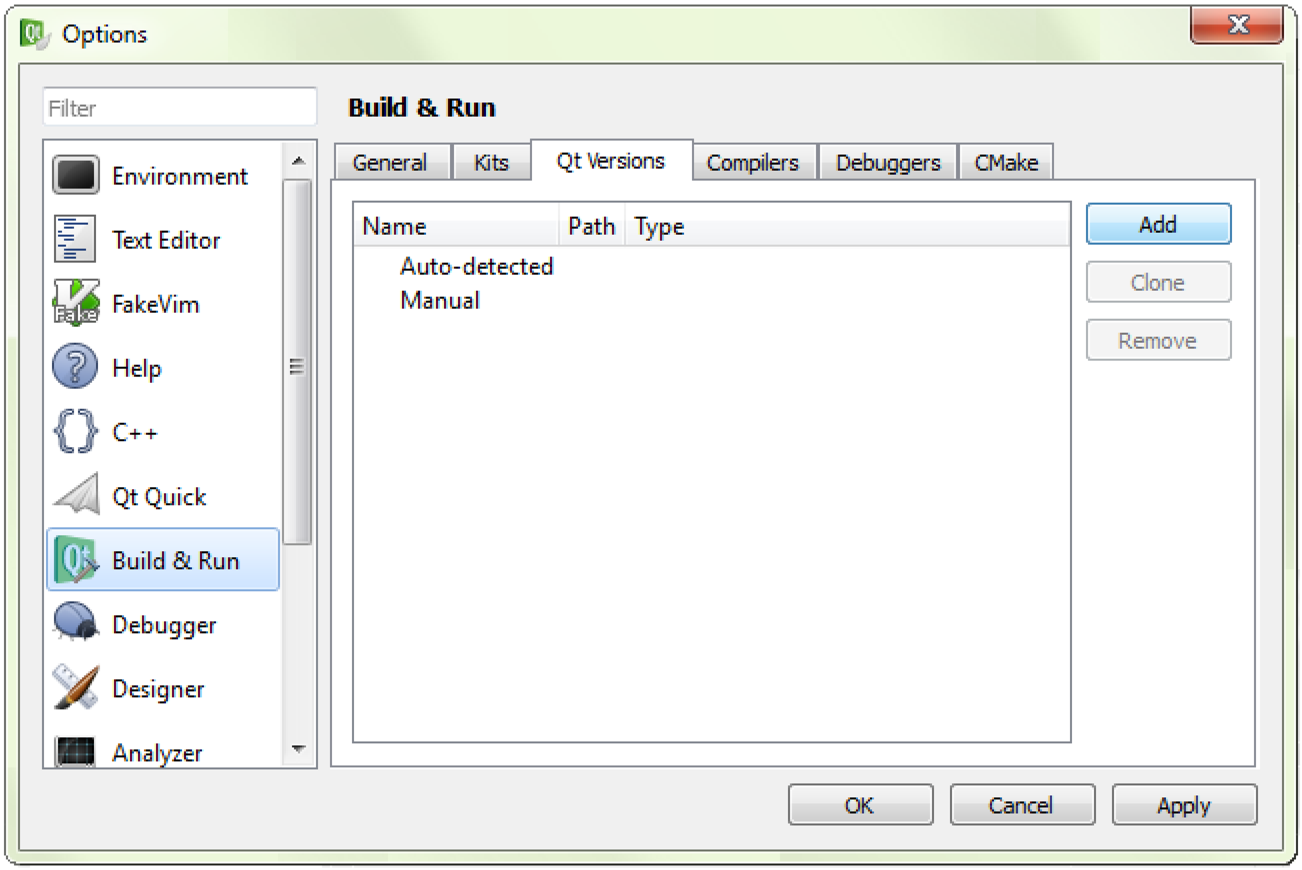
Name: QNX SDP 6.6 QCC

Compiler path: C:\qnx660\host\win32\x86\usr\bin\qcc.exe

NDK/SDP path: C:\qnx660

ABI: arm-linux-generic-elf-32bit arm - linux - generic - elf - 32bit

8. Click the **Apply** button in the bottom right corner to save these settings.
9. Click the **Debuggers** tab, then click the **Add** button on the right side.

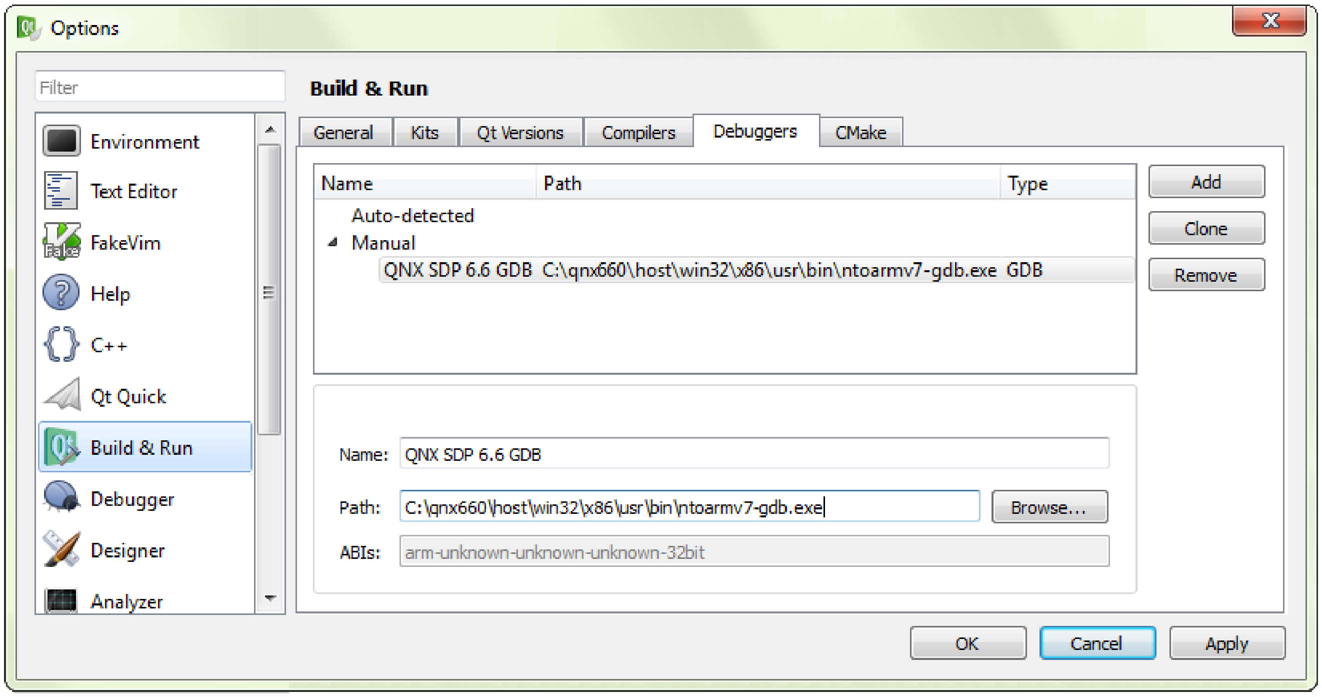


The **Options** dialog displays additional fields at the bottom for configuring a new debugger.

10. Fill in the debugger fields:

    a. In the **Name** field, enter `QNX SDP 6.6 GDB`.

    b. On the `Path` line, click **Browse...** to open the file selector. On Windows, navigate to *DEFAULT_SDP_PATH*`\host\win32\x86\usr\bin` and choose `ntoarmv7-gdb.exe`. On Linux, navigate to *DEFAULT_SDP_PATH*`/host/linux/x86/usr/bin` and choose `ntoarmv7-gdb`. Click **Select Folder** to confirm the setting.



Name	Path	Type
Auto-detected		
Manual		
QNX SDP 6.6 GDB	C:\qnx660\host\win32\x86\usr\bin\ntoarmv7-gdb.exe	GDB

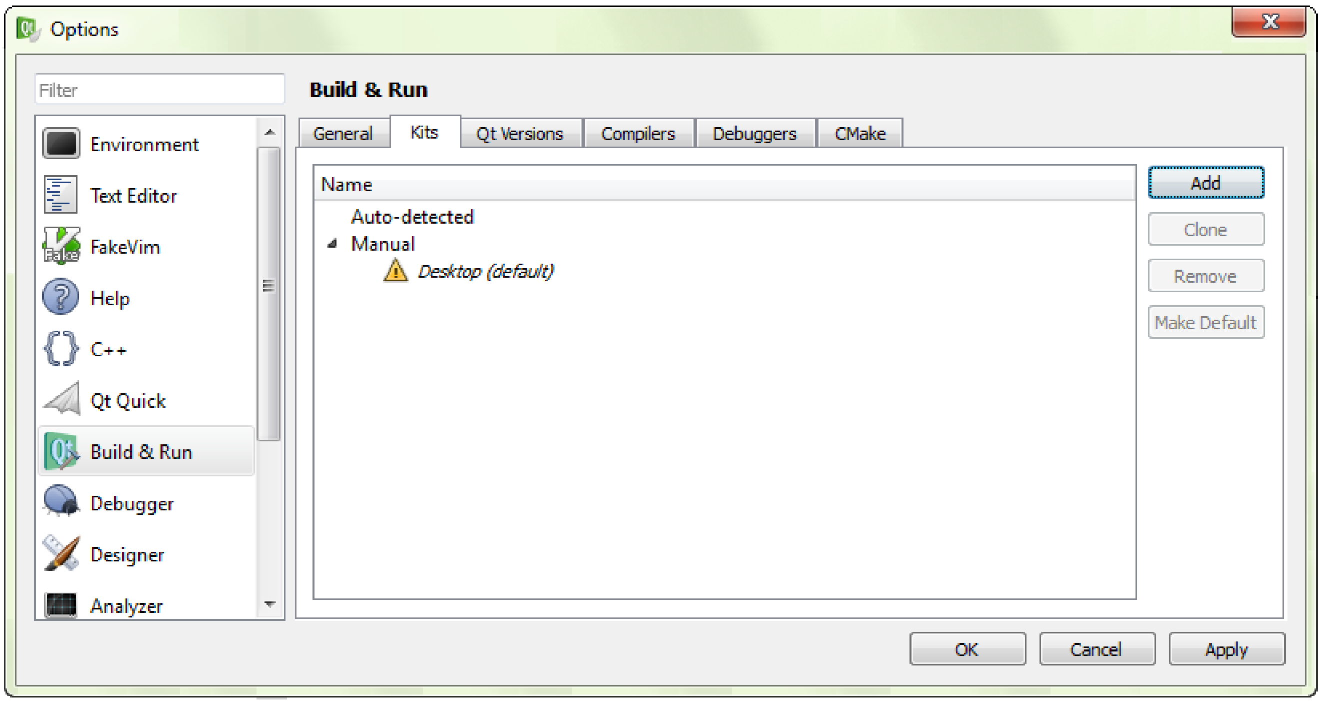
Name: QNX SDP 6.6 GDB

Path: C:\qnx660\host\win32\x86\usr\bin\ntoarmv7-gdb.exe

ABIs: arm-unknown-unknown-unknown-32bit

11. Click the **Apply** button in the bottom right corner to save these settings.

12. Click the **Kits** tab, then click the **Add** button on the right side.



The **Options** dialog displays additional fields at the bottom for configuring a new kit.

13. Fill in the kits fields:

   a. Name the kit something meaningful, like `QNX SDP 6.6 – OMAP5432`.

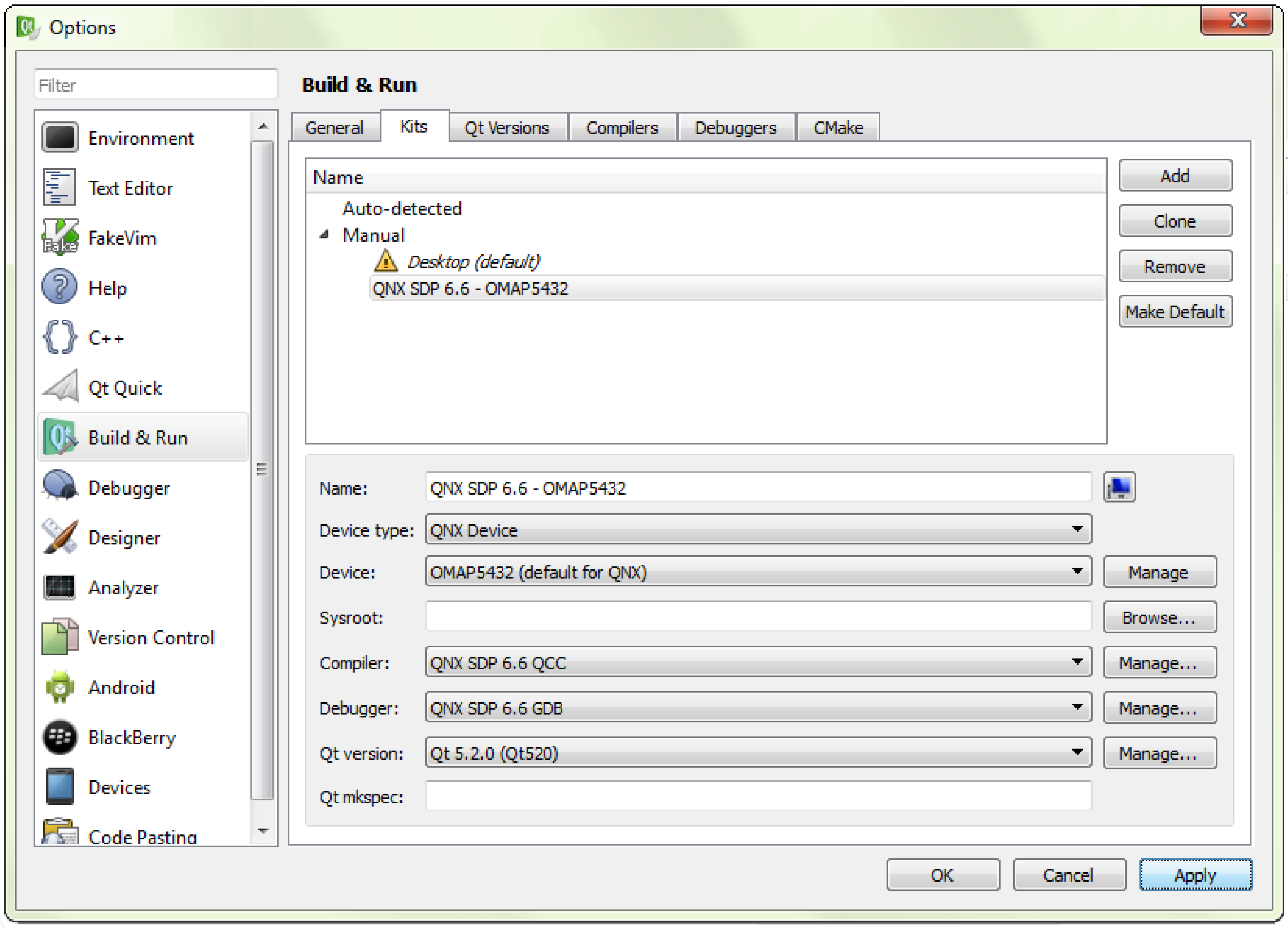
   b. In the **Device Type** dropdown list, select `QNX Device`.

   c. In the **Device** dropdown list, select the device configured earlier (e.g., `OMAP5432`).

   d. In the **Compiler** dropdown list, select `QNX SDP 6.6 QCC`.

   e. In the **Debugger** dropdown list, select `QNX SDP 6.6 GDB`.

   f. In the **Qt version** dropdown list, select `Qt 5.2.0 (Qt520)`.



Name: QNX SDP 6.6 - OMAP5432
Device type: QNX Device
Device: OMAP5432 (default for QNX)
Sysroot:
Compiler: QNX SDP 6.6 QCC
Debugger: QNX SDP 6.6 GDB
Qt version: Qt 5.2.0 (Qt520)
Qt mkspec:

14. Click the **OK** button in the bottom right corner to save all the **Build & Run** settings.

After you've configured a QNX device and a toolchain, you can begin developing Qt apps for QNX CAR 2.1! When creating Qt apps, you can select your Build & Run Kit in the **New Project** wizard to use the build and run settings that you configured earlier.

Chapter 3
Creating and running Qt apps in QNX CAR systems

Qt Creator supports the entire Qt app lifecycle, from creating projects to defining source files and other resources to deploying the app on a target QNX CAR system. After it's installed on your target, you can run the app by tapping its icon in the **Apps Section** screen.

The sections that follow provide a walkthrough of writing a Qt app, packaging it, deploying it on a QNX CAR system, and then running it. Here, *app* refers to a Qt program packaged as a BAR file, which makes it visible in the **Apps Section** screen of the HMI. The steps for writing a more elaborate application (e.g., a new HMI) are the same except for the packaging (because the application would not be packaged as a BAR file).

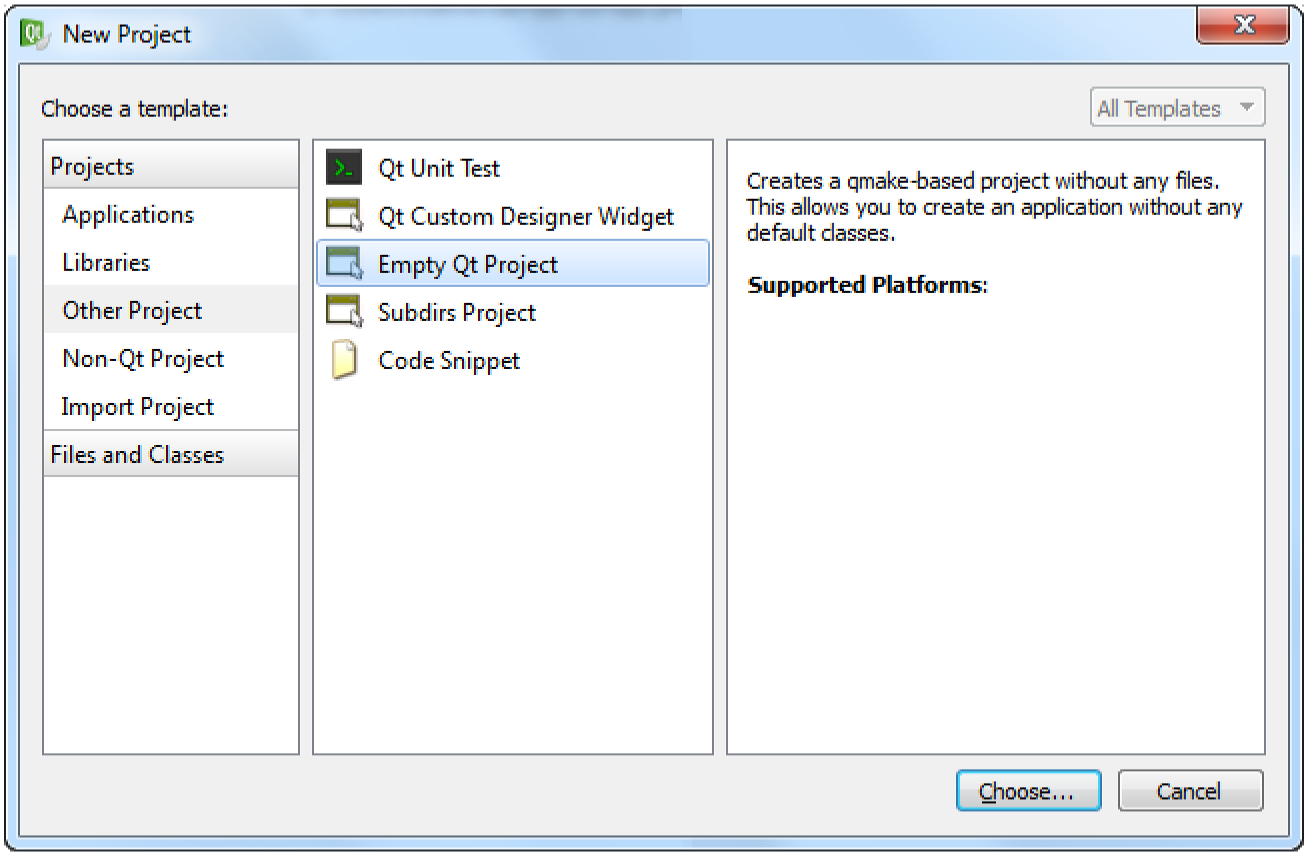
Creating a project for a new Qt App

The first step in creating a Qt App is to create a project in Qt Creator.

This section and the sections that follow show you how to write, package, and deploy a "Hello World" application that can be displayed in the **Apps Section** screen of the QNX CAR HMI. You must have QNX QDF and Qt Creator installed before you can create such Qt apps; for instructions on installing and configuring these components, see "*Preparing your host system for Qt development* (p. 15)".

To create a Qt project:

1. Launch Qt Creator.
2. In the **File** menu, choose **New File or Project...**
3. In the **Projects** dialog, choose **Other Projects**, then **Empty Qt Project**, and then click **Choose...**

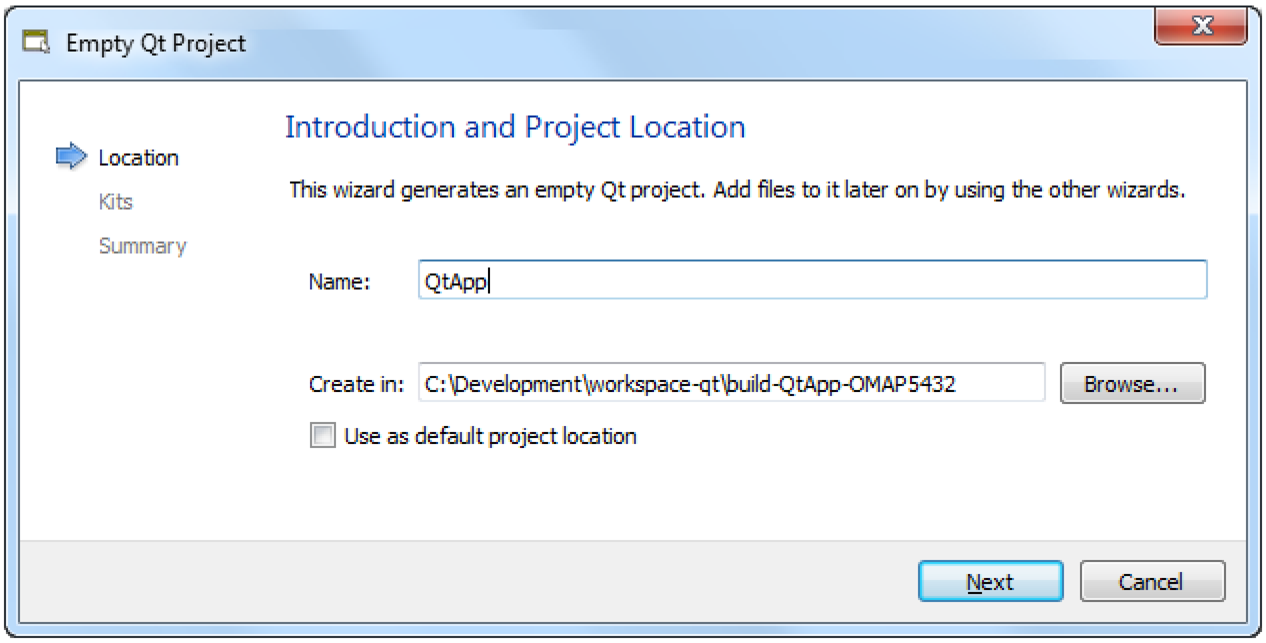


Creates a qmake-based project without any files. This allows you to create an application without any default classes.

Supported Platforms:

Qt Creator displays the **Empty Qt Project** configuration dialog.

4. In the **Location** page, name the project `QtApp`, then click **Next**.


Empty Qt Project

Introduction and Project Location

This wizard generates an empty Qt project. Add files to it later on by using the other wizards.

Name: QtApp

Create in: C:\Development\workspace-qt\build-QtApp-OMAP5432

Use as default project location

5. In the **Kits** page, choose the kit that you configured when setting up Qt Creator (e.g., `QNX SDP 6.6 – OMAP5432`), then click **Next**.

   For details on defining a kit (which specifies toolchain settings), see Step *13* (p. 32) in “Configuring a toolchain in Qt Creator”.

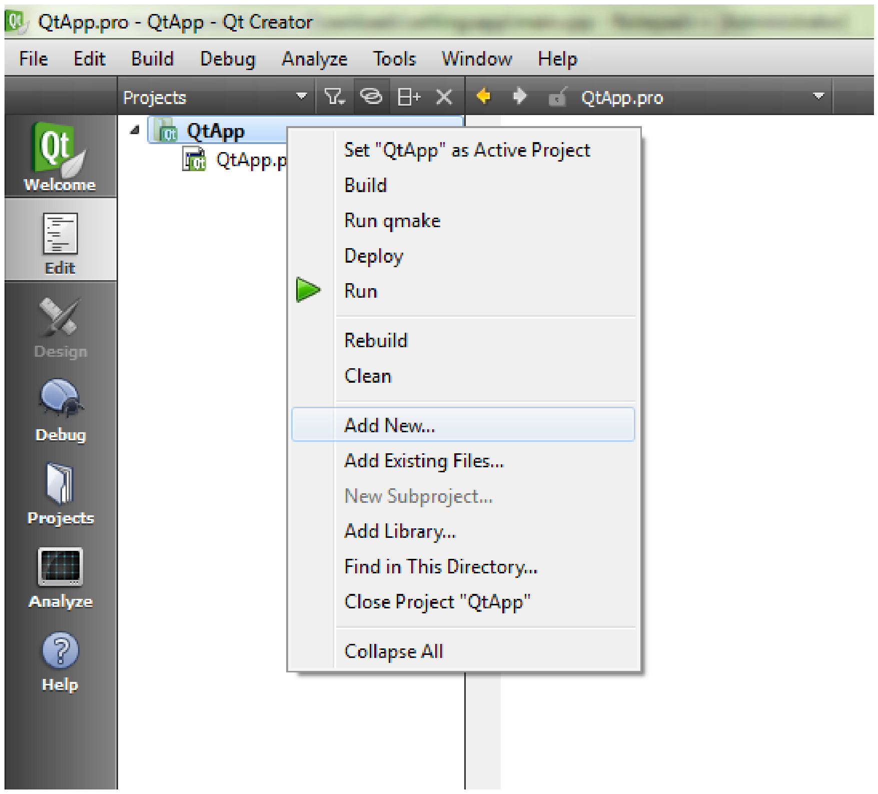
6. In the **Summary** page, click **Finish** to save your new project's settings.

Defining the user interface

You can define the UI by adding a QML file that declares the UI components to your new project.

To define the UI:

1. Click the **Edit** icon on left side, right-click the `QtApp` folder in the **Projects** view, then choose **Add New...**



2. In the **New File** dialog, select `Qt` in the **Files and Classes** list, then `QML File (Qt Quick 2.0)` in the list of specific file types (shown in the middle), then click **Choose...**

   Qt Creator displays the **New QML File (Qt Quick 2.0)** configuration dialog.

3. In the **Location** page, name the file `main`, then click **Next**.
4. In the **Summary** page, click **Finish**.

   The `main.qml` file is opened for editing.

5. Delete the default file content and replace it with the following:

```
import QtQuick 2.0

Rectangle {
    width: 360
    height: 360
```

```
        Text {
            text: qsTr("Hello World")
            anchors.centerIn: parent
        }
    }
```

This QML code defines a simple UI consisting of a square box displaying `Hello World`.

6. Save the file.

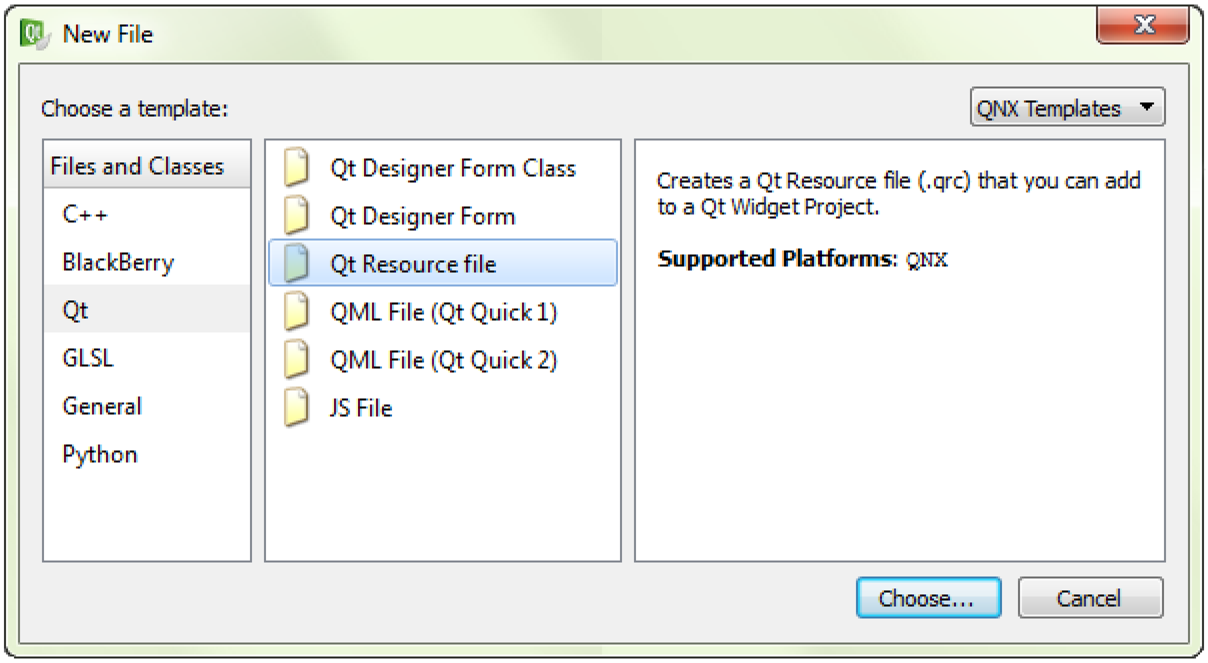
Making a QML file into a project resource

You can create a Qt resource file that includes the QML file that defines the UI. After you add this resource file to your project, Qt Creator will include the UI definition in the binary file.

There are several ways to access resources in Qt apps running on a QNX CAR system. In addition to compiling resources into their binaries, apps can access resources from within their Blackberry ARchive (BAR) file package or from a shared location on the target. It's also possible to use a mix of any of these options. The best solution depends on the nature of the app.

To make the UI-defining QML file into a project resource:

1. Click the **Edit** icon on left side, right-click the `QtApp` folder in the **Projects** view, then choose **Add New...**
2. In the **New File** dialog, select `Qt` in the **Files and Classes** list, then `Qt Resource file` in the list of specific file types (shown in the middle), then click **Choose...**



Creates a Qt Resource file (.qrc) that you can add to a Qt Widget Project.

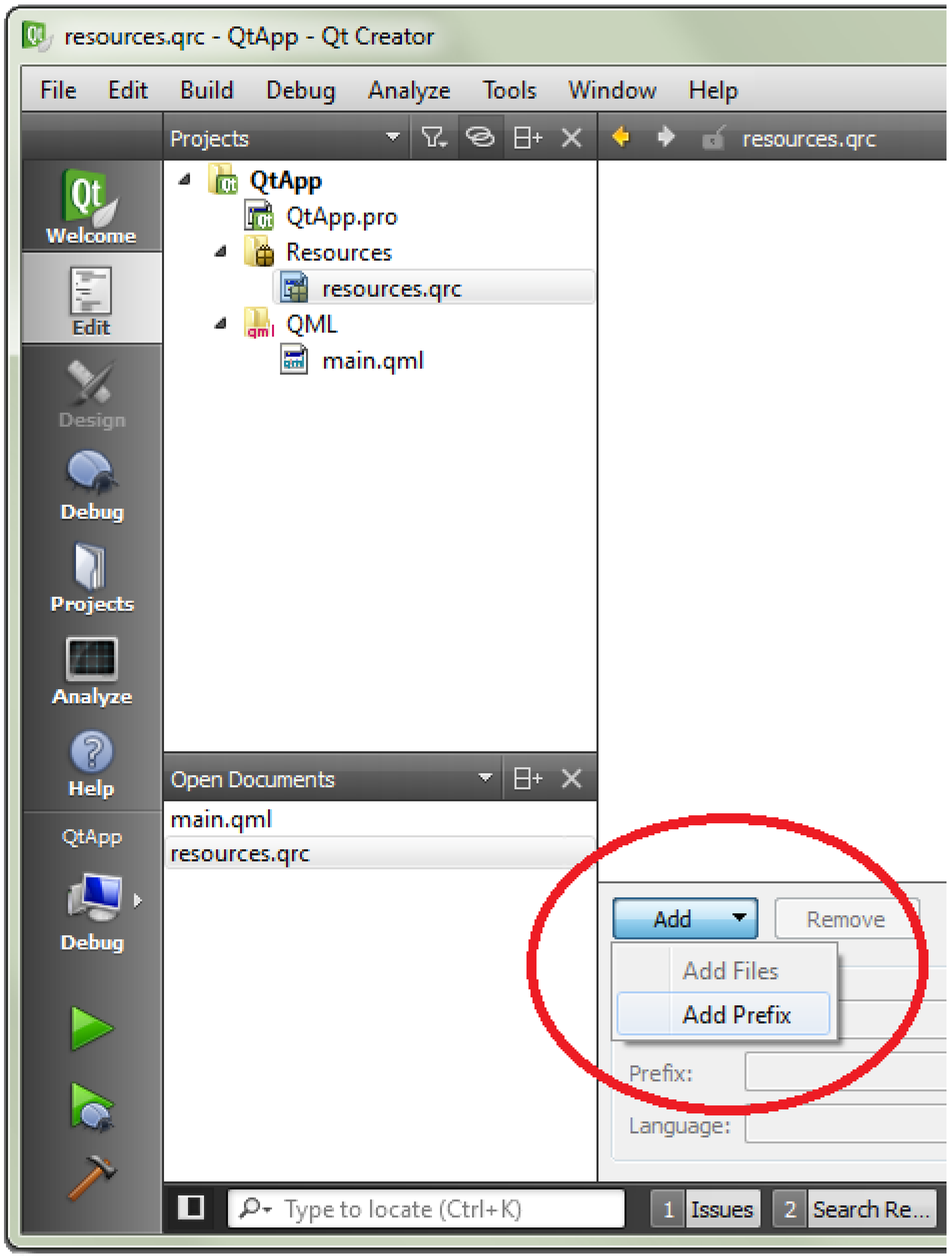
Supported Platforms: QNX

Qt Creator displays the **New Qt Resource file** configuration dialog.

3. In the **Location** page, name the file `resources`, then click **Next**.
4. In the Summary page, click **Finish**.

   A new file, `resources.qrc`, has been added to the project. The **Qt Resources Editor** is open.

5. In the **Projects** view, select the `resources.qrc` file, then click **Add** in the configuration area near the bottom, then choose **Add Prefix**.



6. In the **Prefix** field, replace the default text with `ui`.
7. Click **Add** again, then choose **Add Files**.

   Qt Creator opens a file selector so you can navigate to the file you want to include in the resource.
8. Select `main.qml` and click **Open**.

The `main.qml` file is stored in a Qt resource (`.qrc`) file, which means Qt Creator will compile the QML file into the app binary file.

Adding code to load the UI

The QML file defines how the UI looks but to display it when the Qt app starts, your app must contain C++ source code that defines the application entry point and loads the UI.

To add code that loads the UI:

1. In the **Project** view, right-click the `QtApp` folder and click **Add New...**
2. In the **New File** dialog, select `C++` in the **Files and Classes** list, then `C++ Source file` in the list of specific file types (shown in the middle), then click **Choose...**
3. In the **Location** page, name the file `main`, then click **Next**.
4. In the **Summary** page, click **Finish**.

   The `main.cpp` file is opened for editing.

5. Copy and paste the following code into `main.cpp`:

   ```
   #include <QtGui/QGuiApplication>
   #include <QtQuick/QQuickView>

   int main(int argc, char *argv[])
   {
       QGuiApplication app(argc, argv);

       QQuickView view;
       view.setSource(QUrl("qrc:/ui/main.qml"));
       view.show();

       return app.exec();
   }
   ```

   In this code, the view loads the `main.qml` resource from the Qt resource file, and then displays the UI. Note the syntax for accessing resources in a `.qrc` file, which consists of the resource path prepended with `qrc:`. So, to access `main.qml`, the view uses `qrc:/ui/main.qml` (because the prefix was defined as `ui`).

6. Open the `QtApp.pro` file for editing and add the following line at the end:

   ```
   QT += quick
   ```

   Because `main.cpp` includes the `QtQuick/QQuickView` header file, you must tell Qt Creator to use the `quick` package.

7. Build the project by accessing the **Build** menu and clicking **Build Project "QtApp"**.

Adding an image for the app icon

You can add an icon to your app by saving an image file in your project folder.

To add an image to use as the app icon:

1. Download the following image and save it as `icon.png` in the `QtApp` project folder:

   You can obtain this image from this location on Digia's website:

   http://qt.digia.com/About-Us/Logos-for-Download/

   You can also use any other appropriately sized image file as an icon. We suggest the Qt logo image only for simplicity.

   The icon gets packaged into the app's BAR file—it shouldn't be compiled into `resources.qrc`.

Writing the app descriptor file

To deploy a Qt app on a QNX CAR target, you must package the app in a BAR file. The package must contain an *app descriptor file*, which is an XML file specifying various configuration and application settings.

These instructions show how to define an app descriptor file using Qt Creator but you can manually write this file using whatever editing tool you want.

To write an app descriptor file in Qt Creator:

1. Click the **Edit** icon on left side, right-click the `QtApp` folder in the **Projects** view, then choose **Add New...**
2. In the **New File** dialog, select `General` in the **Files and Classes** list, then `Text file` in the list of specific file types (shown in the middle), then click **Choose...**

   Qt Creator displays the **New Text file** configuration dialog.
3. In the **Location** page, name the file `bar-descriptor.xml`, then click **Next**.
4. In the Summary page, click **Finish**.

   The `bar-descriptor.xml` file is opened for editing.
5. Copy and paste the following content into the new file:

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<qnx xmlns="http://www.qnx.com/schemas/application/1.0">
    <name>Qt App</name>
    <description>The Hello World Qt demo app.</description>
    <icon>
        <image>icon.png</image>
    </icon>
    <id>com.mycompany.QtApp</id>
    <versionNumber>1.0.0</versionNumber>
    <buildId>1</buildId>
    <author>My Company Inc.</author>
    <initialWindow>
        <systemChrome>none</systemChrome>
        <transparent>false</transparent>
    </initialWindow>
    <permission system="true">run_native</permission>
    <action system="true">run_native</action>
    <env var="QQNX_PHYSICAL_SCREEN_SIZE" value="150,90"/>
    <asset type="Qnx/Elf" path="QtApp"
              entry="true">QtApp</asset>
</qnx>
```

   The app-descriptor file provides the app name, description, icon file, and other fields that contain authoring information and settings for the initial window. It also sets the required *QQNX_PHYSICAL_SCREEN_SIZE* environment variable, which defines the height and width of the app's display area on the screen. The environment variables are set using `<env>` tags, as shown. Finally, the app descriptor file must also provide the asset information, which includes the path and format of the binary file.

QNX CAR environment variables

You can set environment variables specific to the QNX CAR platform in the app descriptor file. These settings define the app's display area and its resource paths on the target system.

The QNX CAR environment variables are:

QQNX_PHYSICAL_SCREEN_SIZE

(Required for all apps)

Defines the height and width of the app's display area on the screen. These values are specified in millimeters, not pixels. This is because QNX CAR requires a physical unit and not a virtual unit.

QNXCAR2_ASSETS_DIR

(Optional)

Specifies the path on the target system of any shared resources used by the app. You can share resources among many apps to reduce the sizes of individual apps; this is particularly useful for large resources (e.g., font files). Also, shared resources don't have to be packaged into the app's BAR file or compiled into its binary executable.

LD_LIBRARY_PATH

(Optional)

Specifies the path on the target system of any external libraries (i.e., libraries outside of the Qt framework) used by the app. All QNX Qt development libraries (e.g., `QtQnxCar2`) are located in `/qtcar/lib/`. You can install third-party libraries to this same location or to another location. In the latter case, you need to add the directory that stores these other libraries to the *LD_LIBRARY_PATH* variable, using a colon (`:`) to separate the different entries.

In the app descriptor file, environment variables are set using `<env>` tags, where the `var` attribute lists the variable's name and the `value` attribute lists its value. For instance, the following XML element sets the shared resources path:

```
<env var="QNXCAR2_ASSETS_DIR" value="qtcar/share"/>
```

XML elements in app descriptor file

The app descriptor file must specify the app ID, build ID, version number, a Qt binary file for the entry point, and the physical size of the display area. The file can also define fields such as an icon image file, author name, app name and description, and more.

Name	Required	Description	Attributes	Example
`<action>`	Yes	Specifies the actions associated with the invocation target. Actions are strings that identify the operations that your application is registered to handle. For Qt apps, you must include an `<action>` tag with the value `run_native`, to run the app using the OS runtime.	**`system`** *(Required)* Specifies whether the action is a system action and not a user action. For Qt apps, you must set this attribute to `true`.	`<action system="true">run_native</action>`
`<arg>`	No	Defines the arguments for configuring the application when started. The order of the arguments is important because they're presented in the application's command line in the same order listed in the app descriptor file.		`<arg>-b -v</arg>`
`<aspectRatio>`	No	Specifies whether the application displays in landscape or portrait mode. If no value is specified, the application uses the default orientation set by the OS.		`<aspectRatio>landscape</aspectRatio>`
`<asset>`	Yes	Specifies an asset to package in the BAR file. For Qt apps, you must include an `<asset>` tag that names the Qt binary that's the app entry point. Any assets listed on the command line override those specified with this tag. The text of the tag is a path relative to the BAR package root directory. You can also use the `dest` attribute to specify the asset—this is recommended when using nested `<exclude>` and `<include>` elements.	**`defaultexcludes`** When `yes`, apply the exclusion patterns to the directory tree. For the list of exclusion patterns, see the *`<asset>` element* in the *application descriptor file DTD*.	`<asset type="Qnx/Elf" path="QtApp" entry="true">QtApp</asset>`

Name	Required	Description	Attributes	Example
		Unless otherwise noted, the attributes are optional.	**`dest`** The asset's destination path. Typically, the value is the last part of the `path` value (i.e., the filename). **`entry`** When `true`, use the asset to start the application. The default setting is `false`. **`path`** *(Required)* The location of the asset relative to the current working directory of the packager. **`public`** When `true`, store the asset in the public directory of the BAR file, which is readable by other applications. Icon assets should be public. The default setting is `false`. **`type`** The asset type. For Qt binaries, use `Qnx/Elf`.	
`<au thor>`	No	Specifies the author name (typically the company or developer name).		`<author>My Company Inc.</author>`

Name	Required	Description	Attributes	Example
`<autoOrients>`	No	Indicates whether the application automatically reorients its content when the physical orientation of the device changes.		`<autoOrients>false</autoOrients>`
`<buildId>`	Yes, if not using `<buildIdFile>`	Specifies the build identifier, which is an integer between `0` and `65535`. You modify the value when you want the identifier to change.		`<buildId>1</buildId>`
`<buildIdFile>`	No	Names the file that stores the build identifier. This file is located in your application root folder and it stores the build identifier as an integer. The packager tool increments this value each time you build the BAR package. If you use this element, don't include the `<buildId>` element. The default file created by the Momentics IDE is `buildnum`.		`<buildIdFile>buildnum</buildIdFile>`
`<description>`	No	Defines the text to display when the application is installed. You can use nested `<text>` elements to define text for different languages and locales.		`<description>The Hello World Qt demo app.</description>`
`<env>`	Yes	Defines environment variable settings. For Qt apps, you must define the *QQNX_PHYSICAL_SCREEN_SIZE* variable and you can defined others as well, as explained in "*QNX CAR environment variables* (p. 45)".	**`var`** *(Required)* Name of the environment variable. **`value`** *(Required)* Value of the environment variable.	`<env var="QQNX_PHYSICAL_SCREEN_SIZE" value="150,90"/>`

Name	Required	Description	Attributes	Example

`<exclude>`	No	Specifies the files to exclude from the BAR file based on a filter pattern defined in the `name` attribute. This element is nested within the `<asset>` tag.	**`name`**
			(Required)
			The string patterns for filtering files when excluding them. Use the following tokens to match string values:
			• Asterisk(`*`): Matches zero or more characters.
			• Question mark (`?`): Matches one character.
			• Double asterisk (`**`): Matches zero or more directories or folders.
				`<asset path="Device-Debug/FolderA" type="Qnx/Elf" dest="HelloWorldDisplayManaged">`
				`<exclude name="TEMP?.gif" />`
				`</asset>`

`<filter>`	No	Specifies a target filter. This element is nested within the `<invoke-target>` element. For each target, filters must be declared to describe the kinds of unbound invocations that it supports. Each filter defines an action to perform for MIME types that match the filter. Unbound invocations should generally provide an action, but must provide either a MIME type, URI, or both. They may also define properties, which are sent to the invocation target.		See the *`<invoke-target>`* (p. 51) element.
`<icon>`	No	Defines an icon for the app. The path of the icon file is defined in the nested `<image>` tag. If no file is specified, the app doesn't have any default icon but is		See the *`<image>`* (p. 50) element.

Name	Required	Description	Attributes	Example
		represented by an empty spot in the viewing area showing the installed apps.		
`<id>`	Yes	Provides an identifier (50 characters or less) for your app. We recommend using a reverse DNS-style naming convention for the value. The value is the package name in the BAR file.		`<id>com.mycompany.QtApp</id>`

`<image>`	No	Specifies the location of the icon image to use for the app. The value is the path to the image asset (PNG or JPG file) from the application root path. The recommended image size is 86 x 86 or 90 x 90 pixels.
		
		This element is nested within the `<icon>` element.
				`<icon>`
				
				`<image>icon.png</image>`
				
				`</icon>`

`<include>`	No	Specifies the files to include in the BAR file based on a filter pattern defined in the `name` attribute. This element is nested within the `<asset>` tag.	**name**
			
			(Required)
			
			The string patterns for filtering files when including them. Use the following tokens to match string values:
			
			• Asterisk(`*`): Matches zero or more characters.
			• Question mark (`?`): Matches one character.
			• Double asterisk (`**`): Matches zero or more directories or folders.
				`<asset path="Device-Debug/FolderA" type="Qnx/Elf" dest="HelloWorldDisplayManaged">`
				
				`<include name="App?.png" />`
				
				`</asset>`

`<initialWindow>`	No	Contains elements used to override the default properties of the initial app window.		`<initialWindow>`

Name	Required	Description	Attributes	Example
				`<system Chrome>none</system Chrome>` `<transpar ent>false</transpar ent>` `</initialWindow>`

`<in voke-target>`	No	Defines an invocation target, which allows one application (the target) to be launched from another application (the client). Typically, the target is part of an application or viewer bundled in the BAR file. An application or viewer must declare each of its targets against an entry point (e.g., the `id` attribute value of `<invoke-target>`).
		When declaring a target, a globally unique target-key must be used. To ensure the uniqueness of the value, we recommend that you use conventions such as a reverse-DNS name.
		This element can contain the following elements:
		• `<filter>`
		• `<icon>`
		• `<invoke-target-name>`
		• `<invoke-target-type>`
		• `<splashScreens>`
			`id`
			(Required)
			A globally unique ID that must start with the application package name. The ID can be up to 50 characters.
				`<invoke-target id="com.mycompa ny.pdf.app">`
				`<invoke-target-name>DocFactory</in voke-target-name>`
				`<invoke-target-type>application</in voke-target-type>`
				`<icon>`
				`<image>DocFactoryI con.png</image>`
				`</icon>`
				`<splashScreens>`
				`<image>DocFactoryS plash.png</image>`
				`</splashScreens>`
				`<filter>`
				`<action>bb.ac tion.VIEW</action>`
				`<mime-type>applica tion/pdf</mime-type>`
				`<mime-type>applica tion/x-pdf</mime-type>`
				`<property var="saveOn Close" value="true"/>`

Name	Required	Description	Attributes	Example
				`</filter>` `</invoke-target>`
`<invoke-target-name>`	No	Defines the text to display in the UI. If this value isn't specified, the application name (i.e., the value in `<name>`) will be displayed.		See the *`<invoke-target>`* (p. 51) element.
`<invoke-target-type>`	No	Defines the target type. The supported types are: **application** The target is an application and is started only when required. **viewer** The target reference always spawns a new window and window reparenting is required.		See the *`<invoke-target>`* (p. 51) element.
`<mime-type>`	No	The MIME type of the data that the invocation target can process. This element is nested within the `<filter>` element. The grammar for the MIME type must support these specifications: • RFC 2045 (content-types) • RFC 4288 (IANA registration)		See the *`<invoke-target>`* (p. 51) element.
`<name>`	No	Defines the string value to display when the app is installed. This UTF-8 value can be at most 25 characters.		`<name>Qt App</name>`
`<packageLocale>`	No	Lists the locales supported by the application. The values given must be defined in the IETF Best Current Practice (BCP) 47 specification. You can use a comma-delimited list of locales to specify more than one.		`<packageLocale>en-US,de_DE,fr_CA</packageLocale>`

Name	Required	Description	Attributes	Example
`<permission>`	Yes	Specifies the privileges (also known as capabilities, user actions, or actions) that the application requests from the OS. For Qt apps, you must include a `<permission>` tag with the value `run_native`, to run the app using the OS runtime. The list of permission settings relevant to Qt apps is given in "*App permissions* (p. 55)".	**`system`** *(Required)* Specifies whether the action is a system action and not a user action. For Qt apps, you must set this attribute to `true`.	`<permission system="true">run_native</permission>`
`<platformArchitecture>`	No	Specifies the processor architecture that the application is compiled for. If you don't specify a value, the Momentics IDE inspects the binary to determine the value. You can use the following values: **x86** Specifies to compile your application to run on a simulator. **armle-v7** Specifies to build the application to run on a device.		`<platformArchitecture>x86</platformArchitecture>`
`<platformVersion>`	No	Lists the locales supported by the application. The values given must be defined in the IETF Best Current Practice (BCP) 47 specification. You can use a comma-delimited list of locales to list more than one.		`<platformVersion>10.2.0.155</platformVersion>`
`<property>`	No	Specifies additional arguments to send to an invocation target. This element is nested within the `<filter>` element.	**`var`** *(Required)* Name of the property. **`value`** *(Required)*	See the *`<invoke-target>`* (p. 51) element.

Name	Required	Description	Attributes	Example
			Value of the property.	
`<qnx>`	Yes	Defines the top-level element of the schema used for the app descriptor file.	**`xmlns`** *(Optional)* URL reference to the XML namespace.	See the example of the app descriptor file in "*Writing the app descriptor file* (p. 44)".
`<splash Screens>`	No	Contains `<image>` elements that specify images to display when the application launches. You can nest `<text>` elements within the `<image>` elements to define different splashscreens for different locales. A splashscreen image file must be a PNG or JPG file whose path is in the application root folder.		`<splashScreens>` `<image>` `<text xml:lang="fr">splash-600x1024_fr.jpg</text>` `</image>` `</splashScreens>`
`<splash screen>`	No	Specifies the image file to display when the app is launching. You can use the `<text>` element to specify different images for different languages and locales. This file must be a PNG or JPG file with resolution sizes of 1024 x 600 pixels (landscape) or 600 x 1024 (portrait). The image can be in the application root folder or in a folder accessible from the root.		`<splashscreen>` `sample-splashscreen-landscape.png:sample-splashscreen-por trait.png` `<text xml:lang="de-DE">sample-splash screen-land scapeDE.png:sample-splashscreen-por traitDE.png</text>` `</splashscreen>`
`<system Chrome>`	No	When `standard`, the initial application window is displayed with the standard system chrome (i.e., title bar, borders, and controls) provided by the OS. When `none`, no system chrome is displayed. This setting can't be changed at run time.		See the *`<initialWindow>`* (p. 50) element.

Name	Required	Description	Attributes	Example
`<text>`	No	Specifies text for the parent `<name>` and `<description>` elements, to support different languages and locales. You can also use this element to specify multiple image files for the `<image>` and `<splashscreen>` elements.	**`xml:lang`** *(Required)* The language or locale code. The XML locale strings use hyphens per the IETF Best Current Practice (BCP) 47 specification. For example, you can use `en-US` (United States English), `de-DE` (German), or `fr-CA` (Canadian French).	`<description>The Hello World Qt demo app.` `<text xml:lang="de-DE">The German description for the Hello World Qt demo app.</text>` `</description>`
`<transparent>`	No	Specifies whether the initial application window is alpha-blended with the background. The use of transparency can slow down rendering and consume more memory. The transparency setting can't be changed at run time and is valid only when the `<systemChrome>` element is set to `none`.		See the *`<initialWindow>`* (p. 50) element.
`<versionNumber>`	Yes	Specifies the app version as a string in the format `<0-999>.<0-999>.<0-999>`. The version is useful for determining whether the application requires an upgrade. The value can be a one-, two-, or three-part value, such as `1`, `1.0`, or `1.0.0`.		`<versionNumber>1.0.0</versionNumber>`

App permissions

With the `<permission>` element in the app descriptor file, you can list the permissions you want the OS to grant your application.

The following permissions can be granted:

Functionality or capability	Permission element value	Description
BlackBerry messenger	`bbm_connect`	Allows the app to connect to BlackBerry Messenger (BBM). This permission also allows the app to view contact lists and user profiles, invite BBM contacts to download the app, initiate BBM chats, share content from within the app, and stream data between apps in real time.
Calendar	`access_pimdomain_calendars`	Grants the app access to the calendar. You must set this permission to view, add, and delete calendar appointments.
Camera	`use_camera`	Allows the app to access data from cameras attached to the system. This permission is required to take pictures, record video, and use the camera flash.
Capture Screen	`use_camera_desktop`	Allows the app to take screenshots or videos of the user's screen.
Contacts	`access_pimdomain_contacts`	Grants the app access to the contacts stored on the system. You must set this permission to view, create, and delete contacts.
Control Notification Settings	`access_notify_settings_control`	Allows the app to modify global notification settings. By default, apps have permission to read only their own notification settings. This permission doesn't appear on the **Applications** tab in the IDE. You must add the permission manually on the **Source** tab.
Device Identifying Information	`read_device_identifying_information`	Grants the app access to unique system identifiers such as the PIN and serial number. By setting this permission, you can also access SIM card information.
Email and PIN Messages	`access_pimdomain_messages`	Allows the app to access the email and PIN messages stored on the device. This permission is required to view, create, send, and delete email and PIN messages.
Gamepad	`use_gamepad`	Indicates that the app supports gamepad functionality and that it has official gamepad support in the BlackBerry World storefront.
Internet	`access_internet`	Allows the app to use an Internet connection from a Wi-Fi, wired, or other connection. This permission is required to access a nonlocal destination.
Location	`access_location_services`	Grants the app access to the system's current location and any saved access locations. You must set this permission to access geolocation data, information for geofencing, cell tower information, Wi-Fi data, and Cascades Places.

Functionality or capability	Permission element value	Description
Microphone	`record_audio`	Grants the app access to the audio stream from a microphone attached to the system.
My Contact Info	`read_personally_identifiable_information`	Grants the app access to user information such as the first and last names.
Notebooks	`access_pimdomain_notebooks`	Grants the app access to the content stored in the notebooks on the system. This permission is required to view, add, and delete entries and content from notebooks.
Phone	`access_phone`	Allows the app to determine when a user is on a phone call and to access the phone number assigned to the device and send DTMF (Dual Tone Multi-Frequency) tones.
Phone Control	`control_phone`	Allows the app to combine two calls together, end a call, and use the dial pad during a call.
Post Notifications	`post_notification`	Allows the app to post notifications. This permission doesn't require the user to grant your app access and is granted by the OS when requested.
Push	`_sys_use_consumer_push`	Allows the app to use the BlackBerry Push Service with the BlackBerry Internet Service to receive and request push messages. To use these two services together, you must register with BlackBerry. When you register, you will receive a confirmation email containing information that the application needs to receive and request push messages. For more information about registering, see the *Push Service page* on the public BlackBerry website. When using the Push Service with the BlackBerry Enterprise Server or the BlackBerry Device Service, you don't need to register with BlackBerry and you must not set the Push permission for your app.
Run as Active Frame	`run_when_backgrounded`	Allows the app to perform background processing. Without this permission, the app will be stopped when you switch focus to another app. Use this permission sparingly and only when the app must perform background processing. This permission is useful for apps that play music or manage downloads.

Functionality or capability	Permission element value	Description
Run in Background	`_sys_run_headless`	Allows the app to perform tasks in the background for a short period of time without opening the app. This permission doesn't appear on the **Applications** tab in the IDE. You must add the permission manually on the **Source** tab.
Run in Background Continuously	`_sys_headless_nostop`	Allows the app to run in the background at all times. This permission doesn't appear on the **Applications** tab in the IDE. You must add the permission manually on the **Source** tab.
Shared Files	`access_shared`	Allows the app to read and write files shared between all apps. With this permission set, the app can access pictures, music, documents, and other files stored on the local system, at a remote storage provider, on a media card, or in the cloud.
Text Messages	`access_sms_mms`	Grants the app access to text messages stored on the local system. You must set this permission to view, create, send, receive, and delete text messages.
Wi-Fi Connection	`access_wifi_public`	Allows the app to receive Wi-Fi event notifications such as Wi-Fi scan results or changes in the Wi-Fi connection state. This permission also allows limited Wi-Fi control for hotspot aggregator applications that manage network selection and authentication to a Wi-Fi Hotspot. This permission doesn't allow the app to force a connection to a specific network profile when other available networks with a higher priority are configured for the system. It's not necessary to configure this permission if you only want to retrieve or query information about existing Wi-Fi connections. This permission doesn't appear on the **Applications** tab in the IDE. You must add the permission manually on the **Source** tab.

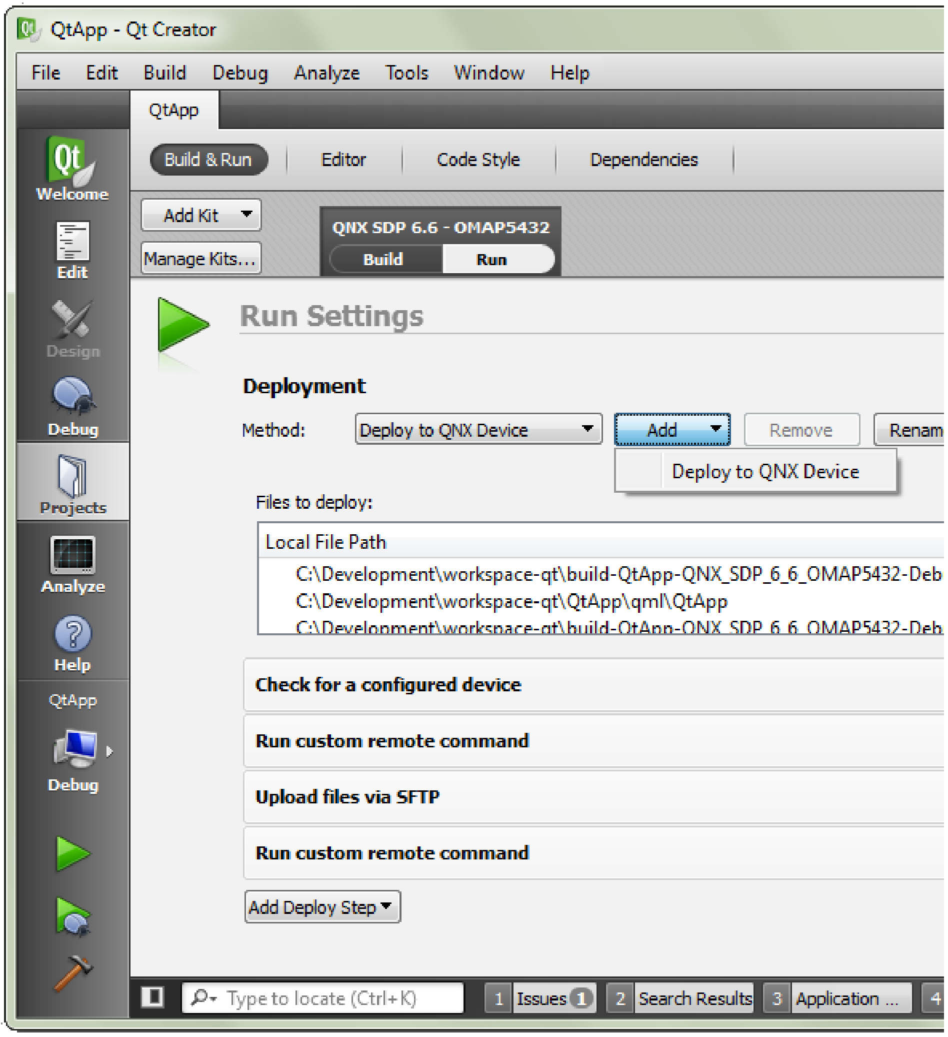
Packaging the app into a BAR file from Qt Creator

After defining the app descriptor file, you can generate a BAR file that contains the app's binary and icon file. The BAR package will be used by the target system to install the app.

These instructions show how to produce a BAR file as a custom build step in Qt Creator, but you can also *generate a BAR file from the command line* (p. 62). BAR files are created by the `blackberry-nativepackager` tool, which is part of the QNX SDK for Apps and Media installation on your host system.

To package the app into a BAR file from Qt Creator:

1. Click the **Projects** icon on left side, select the **Build & Run** tab, click **Add Build Step**, then select `Custom Process Step`:



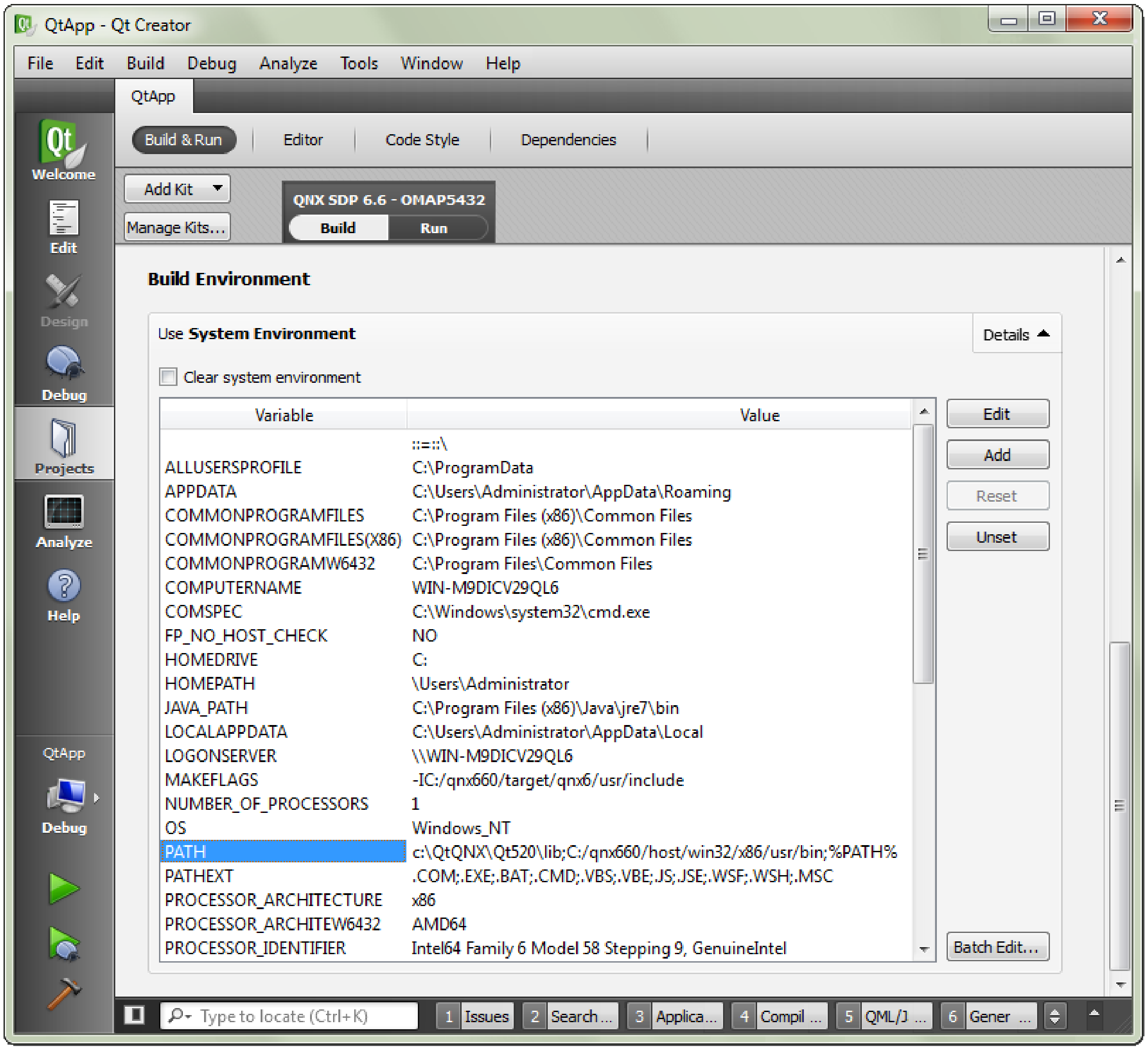
2. On the line that reads **Command**, click **Browse...**.

3. In the file selector dialog, navigate to `DEFAULT_SDP_PATH\host\win32\x86\usr\bin` and choose `blackberry-nativepackager.bat` (on Windows) or navigate to `DEFAULT_SDP_PATH/host/linux/x86/usr/bin` and choose `blackberry-nativepackager` (on Linux).
4. On the line that reads **Arguments**, enter:

   ```
   QtApp.bar %{sourceDir}\bar-descriptor.xml QtApp –C %{sourceDir}
    %{sourceDir}\icon.png
   ```

   These arguments tell the packaging utility to create a file named `QtApp.bar` using the information in `bar-descriptor.xml` and to include `QtApp` (the binary) and `icon.png` in the root folder of the BAR file. For the list of all command options applicable to Qt apps, see "*Qt command-line options for blackberry-nativepackager* (p. 63)".

   This step makes Qt Creator run the `blackberry-nativepackager` command as a build step. Every time you recompile the application, the binary is repackaged into a BAR file.
5. Scroll down to the **Build Environment** section, locate the **Use System Environment** entry, then click **Details** (on the right side).
6. In the list of environment variables, locate *PATH* and if necessary, add the path to the host system's `java.exe` location to the variable's value.



Build Environment

Use **System Environment**

☐ Clear system environment

Variable	Value
	::=::\
ALLUSERSPROFILE	C:\ProgramData
APPDATA	C:\Users\Administrator\AppData\Roaming
COMMONPROGRAMFILES	C:\Program Files (x86)\Common Files
COMMONPROGRAMFILES(X86)	C:\Program Files (x86)\Common Files
COMMONPROGRAMW6432	C:\Program Files\Common Files
COMPUTERNAME	WIN-M9DICV29QL6
COMSPEC	C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK	NO
HOMEDRIVE	C:
HOMEPATH	\Users\Administrator
JAVA_PATH	C:\Program Files (x86)\Java\jre7\bin
LOCALAPPDATA	C:\Users\Administrator\AppData\Local
LOGONSERVER	\\WIN-M9DICV29QL6
MAKEFLAGS	-IC:/qnx660/target/qnx6/usr/include
NUMBER_OF_PROCESSORS	1
OS	Windows_NT
PATH	c:\QtQNX\Qt520\lib;C:/qnx660/host/win32/x86/usr/bin;%PATH%
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	x86
PROCESSOR_ARCHITEW6432	AMD64
PROCESSOR_IDENTIFIER	Intel64 Family 6 Model 58 Stepping 9, GenuineIntel

You can modify the variable's value by clicking the variable name in the display area, clicking **Edit** in the upper right area, and then entering the new value.

The Qt Creator build environment must be configured to find `java.exe` because the `blackberry-nativepackager` command runs a batch file that calls a Java program.

7. Click the **Edit** icon on the left side to return to the editing view, select the **Build** menu, then choose **Build Project "QtApp"**.

   Qt Creator builds the `QtApp` project by compiling the UI-defining QML file into the binary, then generates the BAR file by running the configured packaging command. The IDE displays timestamped messages detailing the outcomes of the build steps in the **Console Output** window.

The QtApp app is packaged in a BAR file and can now be deployed on your target system.

Packaging the BAR file from the command line

You can run the blackberry-nativepackager tool from the command line.

Before running the packaging command, ensure that you have:

- The *app descriptor file*. This XML file must be written manually, whether in Qt Creator or another editor.
- The binary generated by building your Qt app.
- Any resources (statically linked libraries, QML files, icons, etc...) used by the binary. You can compile some resources into the binary or a library linked to the binary. If you choose to do this, you don't need to list those resources on the packaging command line.

The command-line process for packaging a Qt app is similar to the process of "Packaging a native C/C++ app for installation" described in the *Application and Window Management* guide. The key differences are the *QNX CAR environment variables* (p. 45) you can define in the app descriptor file for a Qt app.

To package a Qt app into a BAR file from the command line:

1. In a QNX Neutrino terminal, navigate to the location where your Qt app is stored, then enter the command line to package the app, in this format:

   ```
   blackberry-nativepackager [<commands>] [<options>] bar-package
    app-descriptor binary-file [resource-file]*
   ```

   You must list the BAR file first, followed by the app descriptor file, and then the app files (which must include the binary) to store in the package. Otherwise, the command-line argument order is flexible; you can list the app files in any order and place commands and options at any location in the command line.

   The exact name and location of the packaging tool and its command syntax is platform-dependent. On Linux, the tool is called blackberry-nativepackager and is stored in DEFAULT_SDP_PATH/host/linux/x86/usr/bin/. Any filepaths in the command line must use POSIX notation, using a forward slash (/) to indicate directories. On Windows, it's called blackberry-nativepackager.bat and is stored in DEFAULT_SDP_PATH\host\win32\x86\usr\bin. The command-line filepaths must follow the Windows convention, using a backslash (\) to indicate folders.

   Consider the following packaging command line for a Windows host:

   ```
   blackberry-nativepackager.bat -package AngryBirds.bar -devMode
    birds_bar-descriptor.xml bin/angrybirds a_birds1.png
   ```

This command generates a BAR file named `AngryBirds.bar` based on the `birds_bar-descriptor.xml` file. The BAR file contains the app's binary file (whose path is `bin/angrybirds`) and its icon file (`a_birds1.png`). For details on the `-package` and `-devMode` options and all other command options applicable to packaging Qt apps, see "*Qt command-line options for blackberry-nativepackager* (p. 63)".

After your app is packaged, you can deploy it on the target, as explained in "*Deploying the BAR file on the target* (p. 66)".

Qt command-line options for blackberry-nativepackager

The `blackberry-nativepackager` command line must name the BAR file, app descriptor file, and Qt binary. The packaging tool allows you to list other files to include in the package and supports many command-line options for Qt apps.

Syntax:

```
blackberry-nativepackager [<commands>] [<options>] bar-package
 app-descriptor binary-file [resource-file]*
```

Commands:

`-package`

Package the assets into an unsigned BAR file (this is the default behavior).

`-list`

List all the files in the resulting package. This is useful for debugging packaging issues.

`-listManifest`

Print the BAR manifest. This is useful for debugging.

Packaging options:

`-buildId` ***ID***

Set the build ID (which is the fourth segment of the version). Must be a number from `0` to `65535`.

`-buildIdFile` ***file***

Set the build ID from an existing file and save a new, incremented version to the same file.

`-devMode`

Package the BAR file in development mode. This is required to run unsigned applications and to access application data remotely.

Path options:

`-C` ***dir***

Use *dir* as a root directory. All files listed after this option will be used with tail paths in the output package.

`-e` ***file path***

Save a *file* to the specified *path* in the package.

Other options:

`-version`

Print the packaging tool version.

`help-advanced`

Print the advanced options.

`-help`

Print the usage information. This will include other command-line options and commands that aren't listed here but don't apply to Qt apps.

Variables:

bar-package

Path of the output BAR package file.

app-descriptor

Path of the app descriptor file.

binary-file

Path of the Qt binary file.

resource-file

Path of a resource file used by the Qt app. This could be an icon, a font definition file, an image, and so on. You can name as many resource files as you want.

These paths can be absolute or relative to the current directory. The resulting location in the package is a tail path of the file, unless overridden by the `-C` or `-e` options.

Example:

The command line shown below packages the Settings app from the Qt5 HMI. The app binary, icon file, and several images from installed UI themes are included in the BAR file (`QtSettingsApp.bar`), which is generated based on the app descriptor file (`settings-descriptor.xml`):

```
blackberry-nativepackager.bat -package QtSettingsApp.bar -devMode settings-descriptor.xml
    -e %1\bin\settingsapp bin/settingsapp settings_icon.png
    -C %1\ %1\lib\ %1\share\qnxcar2\palettes\
        %1\share\qnxcar2\fonts\
        %1\share\qnxcar2\qml\main.qml
        %1\share\settingsapp\
        %1\share\qnxcar2\images\themes\720p\default\Settings\
        %1\share\qnxcar2\images\themes\720p\midnightblue\Settings\
        %1\share\qnxcar2\images\themes\800x480\default\Settings\
        %1\share\qnxcar2\images\themes\800x480\midnightblue\Settings\
        %1\share\qnxcar2\images\themes\800x480\titanium\Settings\
        %1\share\qnxcar2\images\themes\720p\default\CommonResources\
        %1\share\qnxcar2\images\themes\720p\midnightblue\CommonResources\
        %1\share\qnxcar2\images\themes\800x480\default\CommonResources\
        %1\share\qnxcar2\images\themes\800x480\midnightblue\CommonResources\
        %1\share\qnxcar2\images\themes\800x480\titanium\CommonResources\
```

In the actual command line, `%1` is replaced with the path of the source directory containing the compiled Qt code. The `-e` and `-C` options take arguments, so the command-line tokens following these options refer to the files affected by them. Here, the `-e` option tells the packaging tool to store the app binary (which is located at `%1\bin\settingsapp` on the host system) at `bin/settingsapp` in the output package. The `-C` option removes the `%1` folder from the paths of the subsequently named files. For example, the files in `%1\lib` on the host system get placed in `/lib` in the package.

Deploying the BAR file on the target

Before you can run an app on the target system, you must copy the app's BAR file to a temporary location on the target and then run the installation script to set up the app. You can configure Qt Creator to automate deploying the BAR file and installing the app.

The steps shown here define commands for Qt Creator to issue to the target as part of the deployment process, automating part of the app development process for convenience. You could also issue these commands manually through a QNX Neutrino terminal connected to the target and the result would be the same.

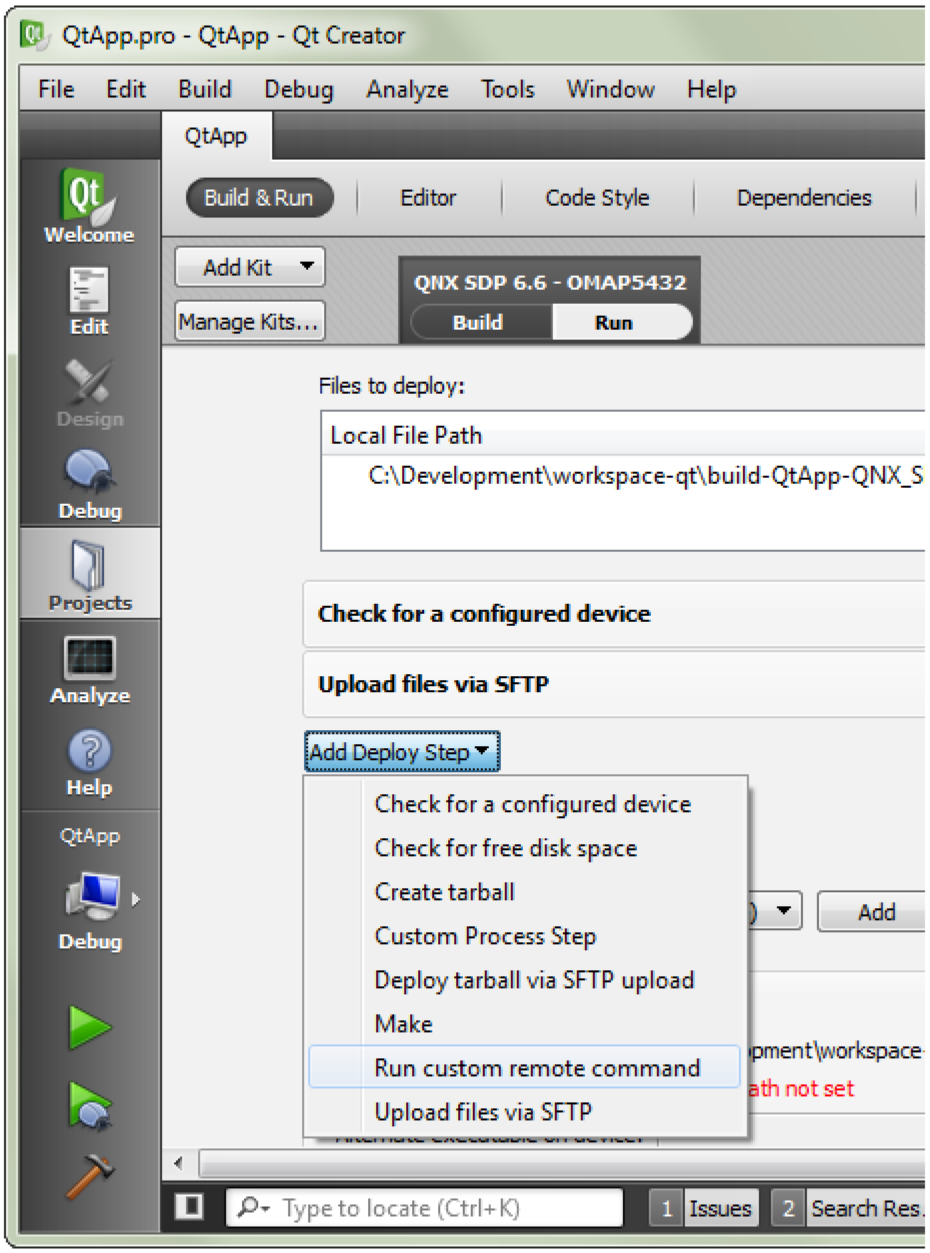
To deploy an app on the target from Qt Creator:

1. Open the project file (`QtApp.pro`) for editing and add the following lines to the end:

   ```
   barfile.path = /var/tmp
   barfile.files = $$OUT_PWD/QtApp.bar
   INSTALLS += barfile
   ```

   This addition to the `INSTALLS` command instructs Qt Creator to copy `QtApp.bar` to `/var/tmp` on the target. The target is represented in Qt Creator as a QNX device, as explained in "*Configuring a QNX device in Qt Creator* (p. 19)".

2. Click the **Projects** icon on left side, select the **Build & Run** tab, click the **Add Deploy Step** button, then choose `Run custom remote command`.



Files to deploy:

Local File Path

C:\Development\workspace-qt\build-QtApp-QNX_S

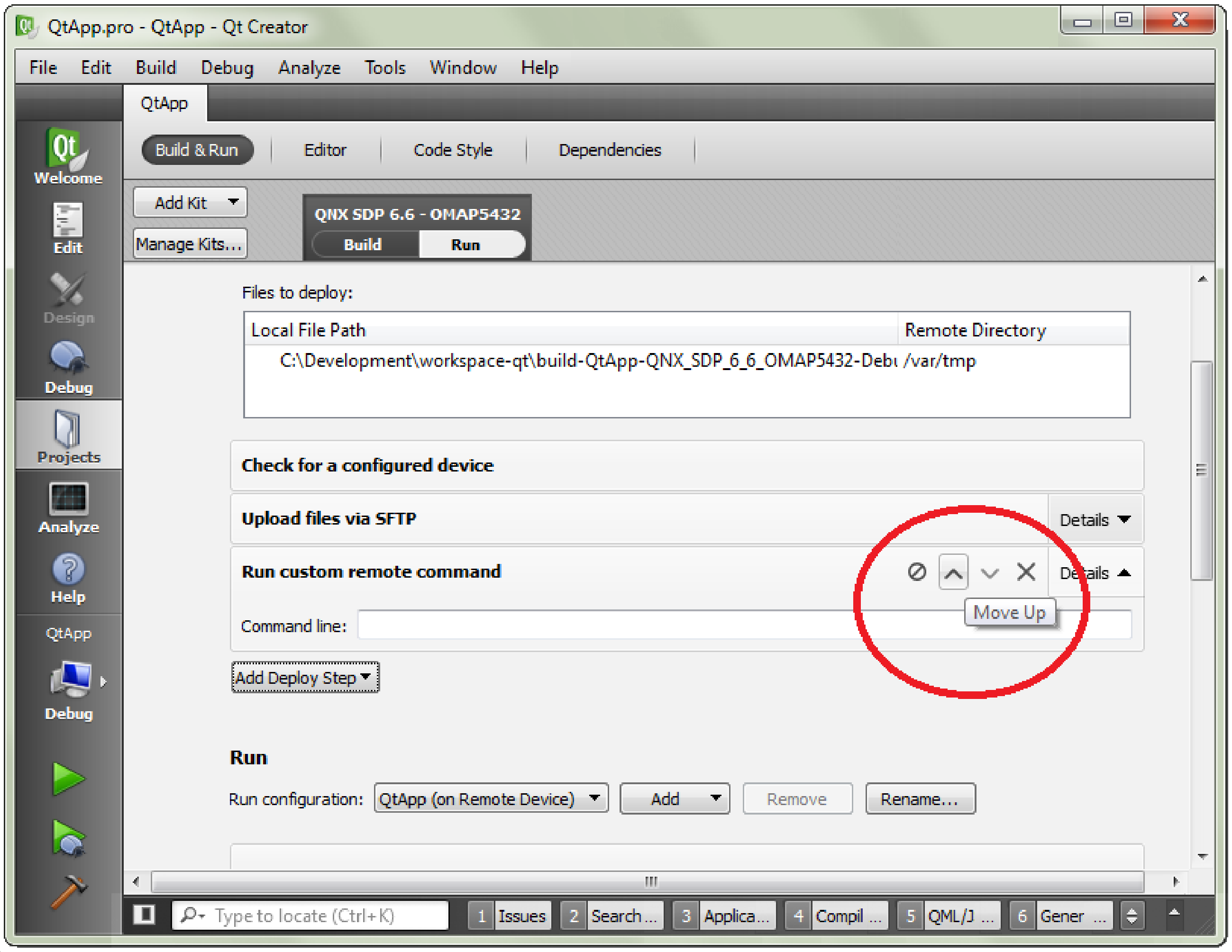
Check for a configured device

Upload files via SFTP

pment\workspace-

ath not set

3. On the line that reads `Run custom remote command`, click the “Move up” button (which has an arrowhead pointing upwards), to ensure that this step is done before the `Upload files via SFTP` step.



Files to deploy:

Local File Path	Remote Directory
C:\Development\workspace-qt\build-QtApp-QNX_SDP_6_6_OMAP5432-Debu	/var/tmp

Command line:

Run configuration: QtApp (on Remote Device)

4. In the **Command Line** text field under `Run custom remote command`, enter the line:

   ```
   mount -uw /base
   ```

   By default, a QNX CAR image has a read-only filesystem. This command makes the filesystem writable, which is necessary to successfully upload files.

5. Click **Add Deploy Step** again, choose `Run custom remote command`, and enter the following command:

   ```
   /base/scripts/bar-install /var/tmp/QtApp.bar
   ```

   This command runs the installer on the target, installing the BAR package in the **Apps** section of the QNX CAR HMI.

   You should now have the following four deployment steps (where the first two were predefined):

   1. Check for a configured device (default)
   2. Run custom remote command: `"mount -uw /base"`
   3. Upload files via SFTP (default)

4. Run custom remote command: `"/base/scripts/bar-install /var/tmp/QtApp.bar"`

6. Click the **Edit** icon on the left side, select the **Build** menu, then choose **Deploy Project "QtApp"**.

   Qt Creator performs the configured deployment steps, first copying the BAR file to the specified target location, and then running the installer script to unpackage the app so it's visible in the **Apps** section. The IDE displays timestamped messages detailing the outcomes of the deployment steps in the **Console Output** window.

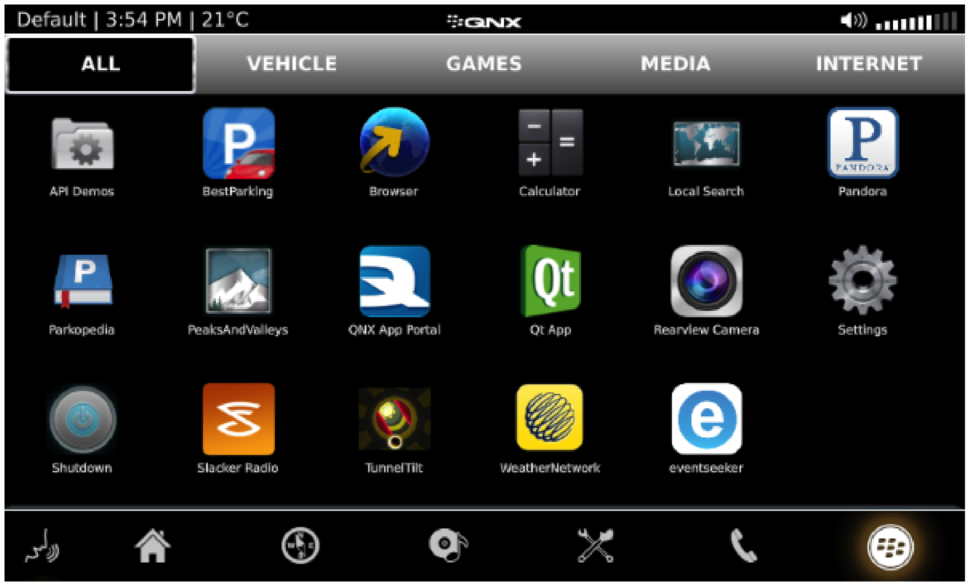
Running the app

After you've unpackaged the app's BAR file on the target, you can run the app from the **Apps Section** screen in the target HMI.

To run the app on the target:

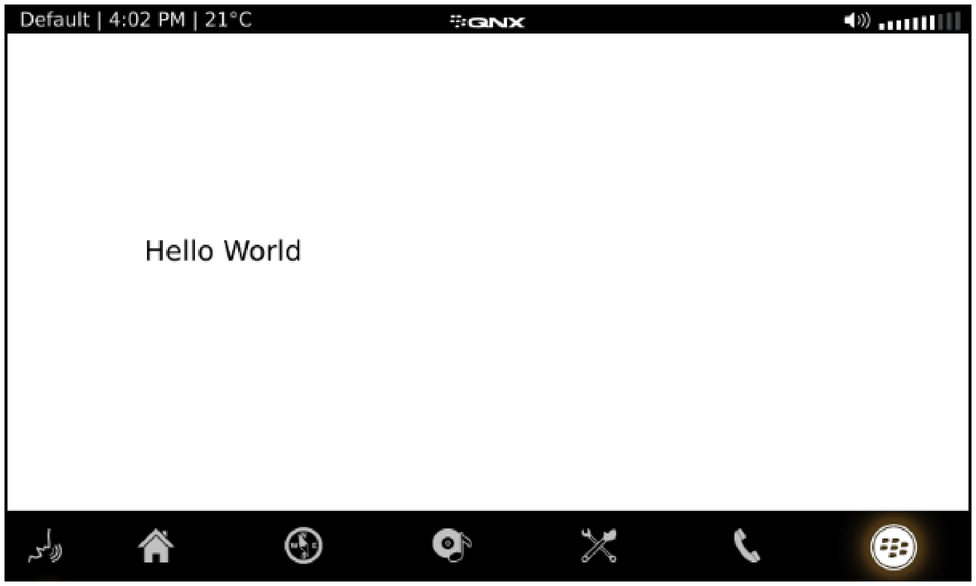
1. Switch to the **Apps Section** screen in the HMI.

   You should see the `QtApp` icon displayed in the list of installed apps:



2. Tap the `QtApp` icon to launch the app.

   `QtApp` launches. You should see the app's basic UI, consisting of the "Hello World" message:



If you specify a splashscreen image with the `<splashscreen>` tag in the app descriptor file, the splashscreen is displayed while the app loads. After it loads, the app displays its initial window based on any properties you configured in the `<initialWindow>` tag, within the physical area defined by the mandatory *QQNX_PHYSICAL_SCREEN_SIZE* environment variable (also set in the app descriptor file).

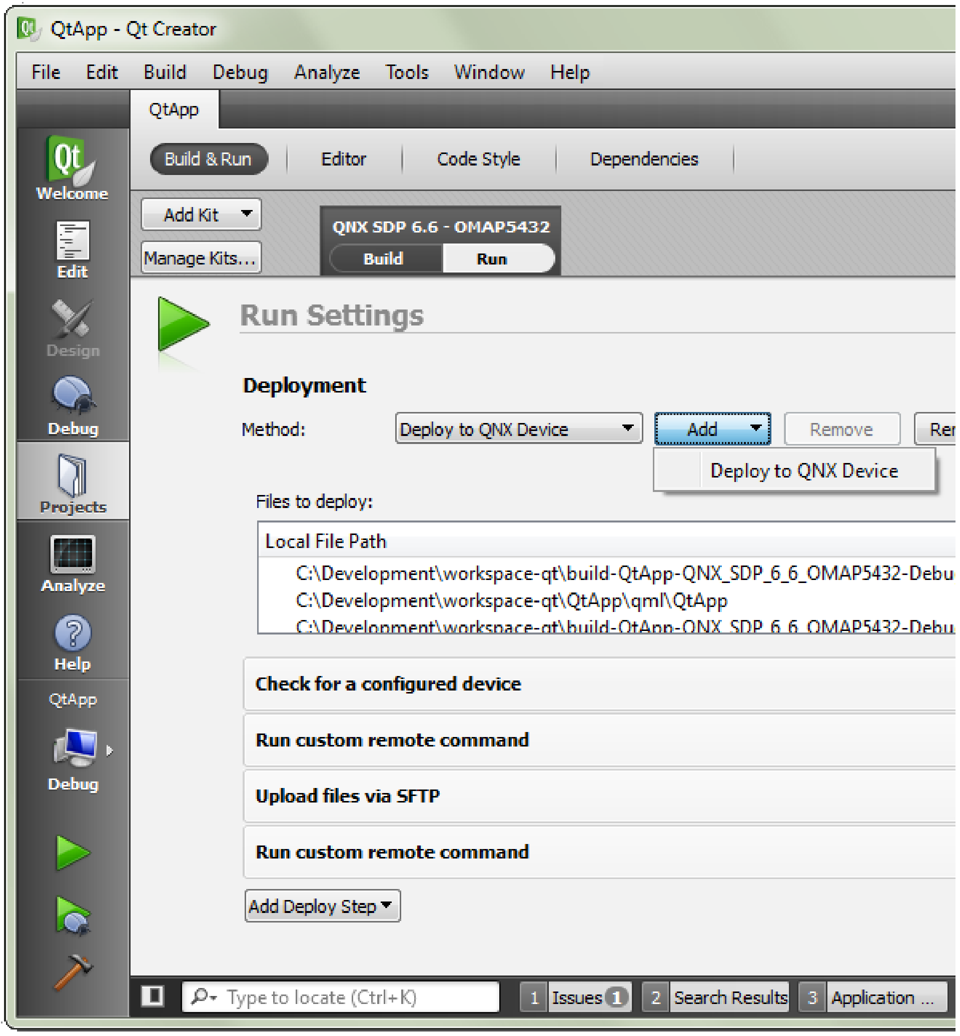
Cleaning the target before redeploying a BAR file

After an app's BAR file has been deployed on the target, we recommend uninstalling the app before redeploying and reinstalling it. You can do this in Qt Creator by creating a second deployment configuration to clean the app's installation on the target.

You can also issue these commands manually through a QNX Neutrino terminal connected to the target and the result will be the same.

To clean an app's installation on the target:

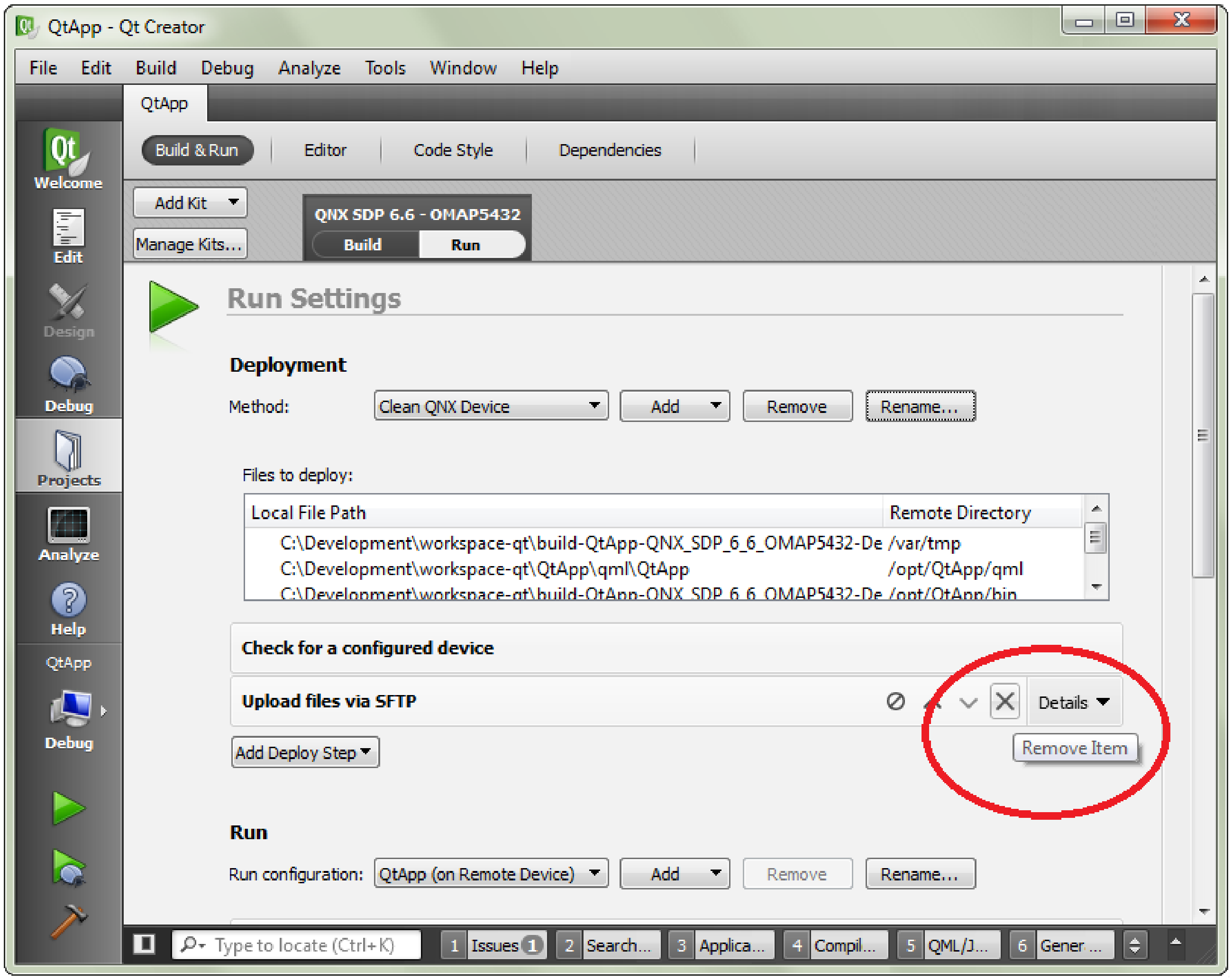
1. Click the **Projects** icon on left side, select the **Build & Run** tab, click the **Add** button in the line that reads **Method**, then choose `Deploy to QNX Device`.



C:\Development\workspace-qt\build-QtApp-QNX_SDP_6_6_OMAP5432-Debu
C:\Development\workspace-qt\QtApp\qml\QtApp
C:\Development\workspace-qt\build-QtApp-QNX_SDP_6_6_OMAP5432-Debu

Check for a configured device
Run custom remote command
Upload files via SFTP
Run custom remote command

2. Click the **Rename...** button on the same line, change the name to `Clean QNX Device`, then click **OK**.
3. Remove the `Upload files via SFTP` step by hovering over the item and clicking the removal button, which is marked with an X.



Run Settings

Deployment

Method: Clean QNX Device

Files to deploy:

Local File Path	Remote Directory
C:\Development\workspace-qt\build-QtApp-QNX_SDP_6_6_OMAP5432-De	/var/tmp
C:\Development\workspace-qt\QtApp\qml\QtApp	/opt/QtApp/qml
C:\Development\workspace-qt\build-QtApp-QNX_SDP_6_6_OMAP5432-De	/opt/QtApp/bin

Check for a configured device

Upload files via SFTP

Run

Run configuration: QtApp (on Remote Device)

4. Click the **Add Deploy Step** button, then choose `Run custom remote command`.
5. In the **Command Line** text field, enter the line:

   `/base/scripts/bar-uninstall com.mycompany.QtApp`

   To uninstall an app, you must provide its ID, which is found in the app descriptor file. For the `QtApp` project, the ID (`com.mycompany.QtApp`) is specified in the fourth element listed inside the root `<qnx>` element in `bar-descriptor.xml`.

There are now two deployment methods. You must choose either `Deploy to QNX Device` or `Clean QNX Device` from the **Method** dropdown menu before running `Deploy Project "QtApp"` in the **Build** menu. To deploy the BAR file and install the app, switch to `Deploy to QNX Device` before running the deployment step. To clean the app's installation on the target, choose `Clean QNX Device` before redeploying the app.

Chapter 4
QPPS API

The `QPPS` library API provides a Qt5 interface to access and update the PPS objects used by QNX CAR platform services. This Qt5 interface replaces the standard PPS interface, which is based on POSIX system calls.

Your apps can create `Object` instances in the `QPPS` API to access the PPS objects used by platform services. When defining an `Object` instance, you must provide the filesystem path of the underlying PPS object that you want to access. Then, you can update and retrieve attributes in that PPS object.

An `Object` emits signals when one attribute has been updated, several attributes have been updated atomically, or the underlying PPS object has been deleted. Each set of atomic attribute updates is stored in a `Changeset` structure.

To accommodate different types of attributes, the `Variant` class lets you store attribute values as strings, numeric types, Booleans, JSON data, or binary data.

The `DirWatcher` class allows you to monitor the PPS objects in a directory. A `DirWatcher` emits signals when objects are added to or removed from the directory.

Finally, the library can be built in *simulator mode*, in which all PPS object reads and writes are handled by the simulator; no filesystem objects are created. This mode is useful on development systems that have no PPS service. You can use the `Object` instances as usual, but also use the `Simulator` object to simulate platform services that add, update, and remove PPS attributes and objects.

All classes in the QPPS API are defined within the `QPps` namespace. If your app uses other classes with names matching the QPPS classes, your code must explicitly refer to the QPPS classes by listing the namespace in front of the class (e.g., `QPps::Object`).

Changeset

All attribute assignments and removals either being made or that have been made to the PPS object.

Synopsis:

```
#include <qpps/changeset.h>

namespace QPps
{
    struct Changeset {
        QMap< QString, Variant > assignments;
        QSet< QString > removals;
    };
}
```

Data:

QMap< QString, Variant > assignments

The attribute assignments.

QSet< QString > removals

The attribute removals.

Library:

`libqpps`

Description:

All attribute assignments and removals either being made or that have been made to the PPS object.

DirWatcher

Watches object additions and removals in a directory.

Synopsis:

```
#include <qpps/dirwatcher.h>

namespace QPps
{
    class QPPS_EXPORT DirWatcher : public QObject
    {
        Q_OBJECT

    public:

        explicit DirWatcher( const QString &path,
                             QObject *parent = 0 );

        ~DirWatcher();

        bool isValid() const;

        QString errorString() const;

        QString path() const;

        QStringList objectNames() const;

    Q_SIGNALS:

        void objectAdded( const QString &name );

        void objectRemoved( const QString &name );

    };
}
```

Library:

`libqpps`

Description:

Watches object additions and removals in a directory. The `DirWatcher` class uses a feature of the PPS service to monitor object additions and removals at specific paths. Also, this class can list the current objects in a particular directory and can signal object additions and removals.

Public functions in `DirWatcher`

Functions defined in the `DirWatcher` class for creating monitors of PPS objects at specific paths, for retrieving the names of the PPS objects and the paths being monitored, and for troubleshooting.

DirWatcher()

Create a `DirWatcher` *to watch object additions and removals in a directory.*

Synopsis:

```
#include <qpps/dirwatcher.h>

DirWatcher( const QString &path,
            QObject *parent = 0 );
```

Arguments:

path

The path of the directory being watched.

parent

A reference to the parent `QObject`. This parameter lets you link a `DirWatcher` to a `QObject` (or a subtype) so the new child object gets deleted when its parent is deleted. By default, no parent is assigned.

Description:

Create a `DirWatcher` to watch object additions and removals in a directory.

~DirWatcher()

Destroy the DirWatcher.

Synopsis:

```
#include <qpps/dirwatcher.h>

~DirWatcher();
```

Description:

Destroy the DirWatcher.

errorString()

Return a string describing the last error, if any.

Synopsis:

```
#include <qpps/dirwatcher.h>

QString errorString() const;
```

Description:

Return a string describing the last error, if any.

Returns:

A description of the last error, if any.

isValid()

Check if the `DirWatcher` *is valid.*

Synopsis:

```
#include <qpps/dirwatcher.h>

bool isValid() const;
```

Description:

Check if the `DirWatcher` is valid.

Returns:

Returns `true` if the `DirWatcher` was opened successfully and no error occurred in the meantime, `false` otherwise.

objectNames()

Return the names of all objects in the watched directory.

Synopsis:

```
#include <qpps/dirwatcher.h>

QStringList objectNames() const;
```

Description:

Return the names of all objects in the watched directory.

Returns:

A `QStringList` containing the names of all objects in the watched directory.

path()

Return the path of the directory being watched.

Synopsis:

```
#include <qpps/dirwatcher.h>

QString path() const;
```

Description:

Return the path of the directory being watched.

Returns:

A `QString` containing the path of the directory being watched.

Signals in `DirWatcher`

Signals emitted by `DirWatcher` objects for indicating object additions and removals.

objectAdded()

Emitted when an object was added to the watched directory.

Synopsis:

```
#include <qpps/dirwatcher.h>

void objectAdded( const QString &name );
```

Arguments:

name

The name of the object added to the directory.

Description:

Emitted when an object was added to the watched directory.

objectRemoved()

Emitted when an object was removed from the watched directory.

Synopsis:

```
#include <qpps/dirwatcher.h>

void objectRemoved( const QString &name );
```

Arguments:

name

The name of the object removed from the directory.

Description:

Emitted when an object was removed from the watched directory.

Object

Represents PPS objects in the `/pps` filepath.

Synopsis:

```
#include <qpps/object.h>

namespace QPps
{
    class QPPS_EXPORT Object : public QObject
    {
        Q_OBJECT

    public:

        enum PublicationMode
        {
            PublishMode = 0,
            SubscribeMode,
            PublishAndSubscribeMode
        };

        explicit Object( const QString &path,
                         PublicationMode mode =
                               PublishAndSubscribeMode,
                         bool create = false,
                         QObject *parent = 0 );

        ~Object();

        bool attributeCacheEnabled() const;

        void setAttributeCacheEnabled( bool cacheEnabled );

        bool isValid() const;

        QString errorString() const;

        QString path() const;

        Variant attribute( const QString &name,
                           const Variant &defaultValue =
                                         Variant() ) const;

        QStringList attributeNames() const;

        bool setAttributes( const Changeset &changes );

    Q_SIGNALS:

        void attributesChanged(
                           const QPps::Changeset &changes );

        void attributeChanged( const QString &name,
                               const QPps::Variant &value );

        void attributeRemoved( const QString &name );
        void objectRemoved();

    public Q_SLOTS:

        bool setAttribute( const QString &name,
                           const QPps::Variant &value );

        bool removeAttribute( const QString &name );

    };
}
```

Library:

`libqpps`

Description:

Represents PPS objects in the `/pps` filepath. The `Object` class communicates with these filesystem objects so you can work with PPS at the level of objects and attributes instead of POSIX system calls.

Each PPS object holds key-value pairs called *attributes*, which store the object's state. By default, the state is saved across reboots. The PPS service provides mechanisms for changing object attributes and sending notifications of attribute changes. The `Object` class uses these mechanisms and provides methods to set one or many attributes at a time and also emits signals containing information on attribute updates or removals.

Public functions in `Object`

Functions defined in the `Object` class for creating data types that access PPS objects, for setting and getting attributes in those objects, for enabling or disabling attribute caching, and for troubleshooting.

attribute()

Return the value of an attribute or a default value if no such attribute is known.

Synopsis:

```
#include <qpps/object.h>

Variant attribute(
            const QString &name,
            const Variant &defaultValue = Variant() ) const;
```

Arguments:

name

The attribute being read.

defaultValue

The default value to return if the attribute isn't found. If you don't define *defaultValue* and the attribute isn't found, an empty `Variant` is returned.

Description:

Return the value of an attribute or a default value if no such attribute is known.

Returns:

Returns a `Variant` containing the attribute's value, when the attribute was found. Otherwise, the `Variant` contains the specified default value or is empty (if no default value was specified).

attributeCacheEnabled()

Get the attribute caching status.

Synopsis:

```
#include <qpps/object.h>
bool attributeCacheEnabled() const;
```

Description:

Get the attribute caching status.

Returns:

The function returns `true` just before an *attributeChanged()* signal is emitted, when attribute names and values are cached internally for later querying through *attributeNames()* and *attribute()*. It returns `false` when no caching takes place. By default, attribute caching is disabled.

attributeNames()

Return the names of all known attributes in the PPS object.

Synopsis:

```
#include <qpps/object.h>
QStringList attributeNames() const;
```

Description:

Return the names of all known attributes in the PPS object.

Returns:

A list of all known attributes.

errorString()

Return a string describing the last error, if any.

Synopsis:

```
#include <qpps/object.h>

QString errorString() const;
```

Description:

Return a string describing the last error, if any.

Returns:

A description of the last error, if any.

isValid()

Check if the PPS object was opened successfully and no error occurred in the meantime.

Synopsis:

```
#include <qpps/object.h>

bool isValid() const;
```

Description:

Check if the PPS object was opened successfully and no error occurred in the meantime.

Returns:

Returns `true` if the PPS object is open and no error has occurred, `false` otherwise.

Object()

Create an `Object` *representing a PPS object.*

Synopsis:

```
#include <qpps/object.h>

explicit Object( const QString &path,
                 PublicationMode mode =
                       PublishAndSubscribeMode,
                 bool create = false,
                 QObject *parent = 0 );
```

Arguments:

path

The path of the PPS object.

mode

The mode of interaction between the application and the PPS object (by default, `PublishAndSubscribeMode`).

create

When `true`, the PPS object will be created if it doesn't exist; when `false`, the object won't be created and an error will be set (if the object doesn't exist). By default, this setting is `false`.

parent

A reference to the parent `QObject`. This parameter lets you link an `Object` to a `QObject` (or a subtype) so the new child object gets deleted when its parent is deleted. By default, no parent is assigned.

Description:

Create an `Object` representing the PPS object located at *path*. The constructor accepts optional arguments for specifying the `PublicationMode` (p. 88), which indicates the direction of information flow between the application and the PPS object, and a create flag, which tells the library whether or not to create the PPS object if it doesn't exist. It also accepts an argument for a parent `QObject`, to link the new `Object` to that other `QObject`.

~Object()

Destroy the Object.

Synopsis:

```
#include <qpps/object.h>

~Object();
```

Description:

Destroy the Object. This method is the destructor for the `Object` class.

path()

Return the path of the underlying PPS object.

Synopsis:

```
#include <qpps/object.h>

QString path() const;
```

Description:

Return the path of the underlying PPS object.

Returns:

The path of the PPS object.

setAttributeCacheEnabled()

Enable or disable attribute caching.

Synopsis:

```
#include <qpps/object.h>

void setAttributeCacheEnabled( bool cacheEnabled );
```

Arguments:

cacheEnabled

When `true`, attribute changes will be cached internally for later querying through *attributeNames()* and *attribute()*. When `false`, no caching will take place *and all previously cached attributes are discarded*.

Description:

Enable or disable attribute caching.

setAttributes()

Write a set of attribute assignments and removals atomically.

Synopsis:

```
#include <qpps/object.h>

bool setAttributes( const Changeset &changes );
```

Arguments:

changes

A `Changeset` (p. 76) listing all requested attribute assignments and removals.

Description:

Write a set of attribute assignments and removals atomically. This way, subscribers won't see a state with only some changes applied.

Returns:

Returns `true` if the PPS object was updated successfully, `false` if an error occurred (call *errorString()* (p. 85) to get more information).

Public properties of `Object`

Properties of the `Object` class for specifying the information flow between the client application and the underlying PPS object.

PublicationMode

Determines which way attribute changes flow between the application and the PPS object.

Synopsis:

```
#include <qpps/object.h>

enum PublicationMode
{
    PublishMode = 0,
    SubscribeMode,
    PublishAndSubscribeMode
};
```

Data:

`PublishMode`

Publish but do not subscribe to attribute changes of the underlying PPS object.

`SubscribeMode`

Subscribe to but do not publish attribute changes of the underlying PPS object.

`PublishAndSubscribeMode`

Publish and subscribe to attribute changes of the underlying PPS object.

When opening a server object in client mode, you must use this setting or you won't properly connect to the object.

Description:

Determines which way attribute changes flow between the application and the PPS object.

Public slots in `Object`

Slots defined in the `Object` class for setting and removing attributes in PPS objects.

removeAttribute()

Remove an attribute from the underlying PPS object.

Synopsis:

```
#include <qpps/object.h>

bool removeAttribute( const QString &name );
```

Arguments:

name

The attribute being removed.

Description:

Remove an attribute from the underlying PPS object.

Returns:

Returns `true` if the PPS object was updated successfully, `false` if an error occurred (call *errorString()* (p. 85) to get more information).

setAttribute()

Set an attribute in the underlying PPS object.

Synopsis:

```
#include <qpps/object.h>

bool setAttribute( const QString &name,
                   const QPps::Variant &value );
```

Arguments:

name

The attribute being set.

value

The attribute's new value.

Description:

Set an attribute in the underlying PPS object.

The PPS service doesn't filter out no-op changes; if an attribute is set, all subscribers will be notified, even if the value didn't change.

Returns:

Returns `true` if the PPS object was updated successfully, `false` if an error occurred (call *errorString()* (p. 85) to get more information).

Signals in `Object`

Signals emitted by `Object` instances for indicating attribute changes and removals from PPS objects as well as removals of PPS objects.

attributeChanged()

Emitted when an attribute in the underlying PPS object has changed.

Synopsis:

```
#include <qpps/object.h>

void attributeChanged( const QString &name,
                       const QPps::Variant &value );
```

Arguments:

name

The name of the attribute that has changed.

value

The attribute's new value.

Description:

Emitted when an attribute in the underlying PPS object has changed.

The PPS service doesn't filter out no-op changes; it may be that attribute *name* was set, but only to its previous value.

attributesChanged()

Emitted when attributes in the underlying PPS object were changed or removed.

Synopsis:

```
#include <qpps/object.h>

void attributesChanged( const QPps::Changeset &changes );
```

Arguments:

changes

A listing of the latest attribute assignments and removals.

Description:

Emitted when attributes in the underlying PPS object were changed or removed. This signal keeps together any set of changes written atomically.

attributeRemoved()

Emitted when an attribute was removed from the underlying PPS object.

Synopsis:

```
#include <qpps/object.h>

void attributeRemoved( const QString &name );
```

Arguments:

name

The name of the attribute that was removed.

Description:

Emitted when an attribute was removed from the underlying PPS object.

objectRemoved()

Emitted when the underlying PPS object was removed.

Synopsis:

```
#include <qpps/object.h>

void objectRemoved();
```

Description:

Emitted when the underlying PPS object was removed. The *isValid()* function will return `false` after this signal has been emitted.

Simulator

Singleton class that simulates a PPS service.

Synopsis:

```
#include <qpps/simulator.h>

namespace QPps
{
    class QPPS_EXPORT Simulator : public QObject
    {
        Q_OBJECT

    public:

        static Simulator* self();

        bool registerClient( const QString &objectPath,
                             QObject *client,
                             QString *errorMessage = 0 );

        void unregisterClient( QObject *client );

        void triggerInitialListing( QObject *client );

        QStringList clientGetAttributeNames(
                                    QObject *client ) const;

        bool clientSetAttribute( QObject *client,
                                 const QString &name,
                                 const QByteArray &value,
                                 const QByteArray &encoding );

        bool clientGetAttribute( QObject *client,
                                 const QString &name,
                                 QByteArray &value,
                                 QByteArray &encoding ) const;

        bool clientRemoveAttribute( QObject *client,
                                    const QString &name );

        void insertAttribute( const QString &objectPath,
                              const QString &key,
                              const QByteArray &value,
                              const QByteArray &encoding );

        void insertObject( const QString &objectPath );

        void reset();

        QMap< QString, QVariantMap > ppsObjects() const;

        void dumpTree( const QString& pathPrefix =
                                               QString() );

    Q_SIGNALS:

        void clientConnected( qulonglong client );
        void clientDisconnected( qulonglong client );

        void attributeChanged( const QString &objectPath,
                               const QString &key,
                               const QByteArray &value,
                               const QByteArray &encoding );

        void attributeRemoved( const QString &objectPath,
                               const QString &key );

        void objectAdded( const QString &objectPath );

    };
}
```

Library:

`libqpps`

Description:

Singleton class that simulates a PPS service.

When QPPS is built in simulator mode, all accesses to PPS objects are redirected to the Simulator object. This object behaves like a real PPS service (with some limitations) and lets you inspect values written to PPS objects and modify those values.

Public functions in `Simulator`

Functions defined in the `Simulator` class for registering clients to receive notifications of updates to simulated PPS objects, for assigning, retrieving, and removing attributes in those objects, and for listing all simulated objects and their contents.

clientGetAttribute()

Get an attribute's value and encoding from the simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

bool clientGetAttribute( QObject *client,
                         const QString &name,
                         QByteArray &value,
                         QByteArray &encoding ) const;
```

Arguments:

client

The client representing the PPS object being read.

name

The name of the attribute being read.

value

A `QByteArray` for storing the attribute's value.

encoding

A `QByteArray` for storing the attribute's encoding.

Description:

Get the value and encoding of the *name* attribute from the PPS object represented by *client*.

Returns:

Returns `true` if the attribute was read successfully, `false` otherwise.

clientGetAttributeNames()

Get the names of all attributes in the simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

QStringList clientGetAttributeNames( QObject *client ) const;
```

Arguments:

client

The client representing the PPS object being read.

Description:

Get the names of all attributes in the simulated PPS object.

Returns:

A list of the names of all attributes in the PPS object.

clientRemoveAttribute()

Remove an attribute from the simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>
```

```
bool clientRemoveAttribute( QObject *client,
                            const QString &name );
```

Arguments:

client

The client representing the PPS object being updated.

name

The name of the attribute being removed.

Description:

Remove the *name* attribute from the PPS object represented by *client*.

Returns:

Returns `true` if the attribute was removed successfully, `false` otherwise.

clientSetAttribute()

Set an attribute in the simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

bool clientSetAttribute( QObject *client,
                         const QString &name,
                         const QByteArray &value,
                         const QByteArray &encoding );
```

Arguments:

client

The client representing the PPS object being updated.

name

The name of the attribute being updated.

value

The new value to assign the attribute.

encoding

The encoding (type) of the attribute. For details on encoding, see the `Object` class.

Description:

Set the *name* attribute to *value* using the specified *encoding*, in the PPS object represented by *client*.

Returns:

Returns `true` if the attribute was written successfully, `false` otherwise.

dumpTree()

Dump the contents of PPS objects to the standard error stream.

Synopsis:

```
#include <qpps/simulator.h>

void dumpTree( const QString& pathPrefix = QString() );
```

Arguments:

pathPrefix

The prefix in the paths of the PPS objects being outputted to `stderr`. An empty prefix means all objects are outputted; this is the default behavior.

Description:

This function outputs any simulated PPS objects with paths starting with *pathPrefix* to the standard error stream (`stderr`).

insertAttribute()

Insert an attribute into a simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

void insertAttribute( const QString &objectPath,
                      const QString &key,
                      const QByteArray &value,
                      const QByteArray &encoding );
```

Arguments:

objectPath

The path of the simulated PPS object.

key

The name of the attribute being inserted.

value

The attribute's value.

encoding

The attribute's encoding.

Description:

Insert an attribute named *key* with the specified *value* and *encoding* into the simulated object at *objectPath*. This function lets you simulate a process writing to a PPS object in the filesystem. The simulation is transparent to the clients that have registered for updates to that object, meaning they can read the inserted attributes as though they were stored in a real PPS object.

insertObject()

Insert a simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

void insertObject( const QString &objectPath );
```

Arguments:

objectPath

The path of the simulated PPS object.

Description:

Insert a simulated PPS object at the path in *objectPath*. No actual filesystem entry is created, but the simulator allows clients to assign and remove attributes in the object as though it were really located at *objectPath*.

ppsObjects()

Get the names of all simulated PPS objects and their attribute settings.

Synopsis:

```
#include <qpps/simulator.h>

QMap< QString, QVariantMap > ppsObjects() const;
```

Description:

Get the names of all simulated PPS objects and the attribute settings of each object. This information is returned in a `QMap`. Each map entry contains a `QString`, which stores an object's name, and a `QVariantMap`, which stores the key-value pairs of all the object's attributes.

Returns:

The names of the PPS objects defined in the simulator along with their attribute settings.

registerClient()

Register a client to receive notifications of updates to a simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

bool registerClient( const QString &objectPath,
                     QObject *client,
                     QString *errorMessage = 0 );
```

Arguments:

objectPath

The path of the simulated PPS object.

client

The client to register for update notifications related to the simulated object at *objectPath*.

errorMessage

A string for storing error messages. By default, no string is defined and error messages are discarded.

Description:

Register the *client* to receive notifications of updates to the simulated PPS object at *objectPath*. All attribute assignments and removals will be handled by the simulator; no PPS object will be created in the filesystem.

Returns:

Returns `true` if the client was registered successfully, `false` if an error occurred.

reset()

Clear all PPS objects.

Synopsis:

```
#include <qpps/simulator.h>

void reset();
```

Description:

This function clears all PPS objects defined in the simulator. These objects still exist but have no attributes after this function is called.

self()

Retrieve the Simulator object.

Synopsis:

```
#include <qpps/simulator.h>

static Simulator* self();
```

Description:

Retrieve a reference to the global `Simulator` object. You must call this function before any others in the `Simulator` class to obtain the object reference, and then use that reference to call the other functions.

Returns:

A reference to the global instance of the simulator.

triggerInitialListing()

Trigger the initial listing of all attributes for a client.

Synopsis:

```
#include <qpps/simulator.h>

void triggerInitialListing( QObject *client );
```

Arguments:

client

The client requesting an initial listing of attributes.

Description:

Trigger the initial listing of all attributes for a client.

unregisterClient()

Unregister a client from receiving notifications of updates to a simulated PPS object.

Synopsis:

```
#include <qpps/simulator.h>

void unregisterClient( QObject *client );
```

Arguments:

client

The client to unregister.

Description:

Unregister a client. This client will no longer receive notifications of updates to the simulated PPS object that it registered when calling *registerClient()* (p. 99).

Signals in `Simulator`

Signals emitted by `Simulator` objects for indicating new client connections and disconnections, attribute updates and removals, and PPS object additions.

attributeChanged()

Emitted when an attribute has been added or updated.

Synopsis:

```
#include <qpps/simulator.h>

void attributeChanged( const QString &objectPath,
                       const QString &key,
                       const QByteArray &value,
                       const QByteArray &encoding );
```

Arguments:

objectPath

The path of the affected PPS object.

key

The name of the attribute that was added or updated.

value

The attribute's new value.

encoding

The attribute's encoding.

Description:

Emitted when an attribute has been added or updated.

attributeRemoved()

Emitted when an attribute has been removed.

Synopsis:

```
#include <qpps/simulator.h>

void attributeRemoved( const QString &objectPath,
                       const QString &key );
```

Arguments:

objectPath

The path of the affected PPS object.

key

The name of the attribute that was removed.

Description:

Emitted when an attribute has been removed.

clientConnected()

Emitted when a client has registered with the simulator.

Synopsis:

```
#include <qpps/simulator.h>

void clientConnected( qulonglong client );
```

Arguments:

client

A unique number identifying the client that has just registered.

Description:

Emitted when a client has registered with the simulator.

clientDisconnected()

Emitted when a client has unregistered with the simulator.

Synopsis:

```
#include <qpps/simulator.h>

void clientDisconnected( qulonglong client );
```

Arguments:

client

A unique number identifying the client that has just unregistered.

Description:

Emitted when a client has unregistered with the simulator.

objectAdded()

Emitted when a new PPS object has been added in the simulator.

Synopsis:

```
#include <qpps/simulator.h>

void objectAdded( const QString &objectPath );
```

Arguments:

objectPath

The path of the newly added PPS object.

Description:

Emitted when a new PPS object has been added in the simulator.

There is no *objectRemoved()* signal because the simulator doesn't support removing objects.

Variant

Stores the value and type information for a PPS attribute.

Synopsis:

```
#include <qpps/variant.h>

namespace QPps
{
    class QPPS_EXPORT Variant
    {
    public:

        Variant();

        Variant( const QByteArray &value,
                 const QByteArray &encoding );

        Variant( const QString &stringValue );

        Variant( const QByteArray &value );

        Variant( bool value );

        Variant( int value );

        Variant( double value );

        Variant( const QJsonObject &value );

        Variant( const QJsonDocument &value );

        bool isValid() const;

        QByteArray value() const;

        QByteArray encoding() const;

        QString toString() const;

        QByteArray toByteArray() const;

        bool toBool() const;

        int toInt( bool *ok = 0 ) const;

        double toDouble( bool *ok = 0 ) const;

        QJsonDocument toJson(
                   QJsonParseError *error = 0 ) const;

        bool operator==( const Variant &other ) const;

        bool operator!=( const Variant &other ) const;

    };
}
```

Library:

`libqpps`

Description:

Stores the value and type information for a PPS attribute. Each attribute consists of a raw value and an encoding, both stored as strings. The encoding indicates the attribute value's "real" type and how to translate between the string representation and the real type.

PPS doesn't standardize any encodings but a few are common in practice:

- `b` for Boolean (`bool`)
- `n` for numbers (floating-point or integer)
- `b64` for binary data (`QByteArray`)
- `json` for JSON format (`QJsonObject` or `QJsonDocument`)
- `s` for string

This class handles all of these encodings.

Public functions in `Variant`

Functions defined in the `Variant` class for constructing attributes of different types, for retrieving the values and encodings of attributes, and for converting attributes to specific data types.

encoding()

Get the attribute's encoding, which indicates how to transform its raw value into a meaningful type.

Synopsis:

```
#include <qpps/variant.h>

QByteArray encoding() const;
```

Description:

Get the attribute's encoding, which indicates how to transform its raw value into a meaningful type. If the encoding is empty, the raw value is probably meant to be used as is.

Returns:

A `QByteArray` containing the attribute's encoding.

isValid()

Check if the `Variant` *is valid.*

Synopsis:

```
#include <qpps/variant.h>

bool isValid() const;
```

Description:

Check if the `Variant` is valid.

Returns:

Returns `true` if the `Variant` is valid, `false` otherwise. For this release, only the default constructor can create an invalid variant.

toBool()

Convert the attribute value to a bool (`b` *encoding).*

Synopsis:

```
#include <qpps/variant.h>

bool toBool() const;
```

Description:

Convert the attribute value to a `bool` (`b` encoding).

Returns:

A `bool` containing the attribute's value.

toByteArray()

Convert the attribute value to a `QByteArray` *(*`b64` *encoding).*

Synopsis:

```
#include <qpps/variant.h>

QByteArray toByteArray() const;
```

Description:

Convert the attribute value to a `QByteArray` (`b64` encoding).

Returns:

A `QByteArray` containing the attribute's value.

toDouble()

Convert the attribute value to a double (`n` encoding).

Synopsis:

```
#include <qpps/variant.h>

int toDouble( bool *ok = 0 ) const;
```

Arguments:

ok

A reference to a `bool` for storing the conversion status. When the reference is non-NULL, the status is set to `false` if a conversion error occurs; otherwise, it's set to `true`.

Description:

Convert the attribute value to a `double` (`n` encoding).

Returns:

A `double` containing the attribute's value.

toInt()

Convert the attribute value to an int (`n` encoding).

Synopsis:

```
#include <qpps/variant.h>

int toInt( bool *ok = 0 ) const;
```

Arguments:

ok

A reference to a `bool` for storing the conversion status. When the reference is non-NULL, the status is set to `false` if a conversion error occurs; otherwise, it's set to `true`.

Description:

Convert the attribute value to an `int` (`n` encoding).

Returns:

An `int` containing the attribute's value.

toJson()

Convert the attribute value to a JSON document (`json` encoding).

Synopsis:

```
#include <qpps/variant.h>

QJsonDocument toJson( QJsonParseError *error = 0 ) const;
```

Arguments:

error

A reference to a `QJsonParseError` object for storing the conversion outcome. When the reference is non-NULL, the object will contain information about any JSON parsing error that may have occurred.

Description:

Convert the attribute value to a JSON document (`json` encoding).

Returns:

A `QJsonDocument` containing the attribute's value.

toString()

Convert the attribute value to a `QString`.

Synopsis:

```
#include <qpps/variant.h>

QString toString() const;
```

Description:

Convert the attribute value to a `QString`. This function is similar to *value()*, except that it checks if the encoding is either empty or `s` (string).

Returns:

A `QString` containing the attribute's value.

value()

Get the attribute's raw value.

Synopsis:

```
#include <qpps/variant.h>

QByteArray value() const;
```

Description:

Get the attribute's raw value.

Returns:

The raw value, as a `QByteArray`.

Variant()

Construct an empty `Variant`*.*

Synopsis:

```
#include <qpps/variant.h>

Variant();
```

Description:

Construct an empty `Variant`. The *isValid()* function will return `false`.

Variant(bool)

Construct a `Variant` *containing a bool (*`b` *encoding).*

Synopsis:

```
#include <qpps/variant.h>

Variant( bool value );
```

Arguments:

value

The Boolean to store.

Description:

Construct a `Variant` containing a `bool` (b encoding).

Variant(double)

Construct a `Variant` *containing a double (n encoding).*

Synopsis:

```
#include <qpps/variant.h>

Variant( double value );
```

Arguments:

value

The double-precision floating-point number to store.

Description:

Construct a `Variant` containing a `double` (n encoding).

Variant(int)

Construct a `Variant` *containing an int (n encoding).*

Synopsis:

```
#include <qpps/variant.h>

Variant( int value );
```

Arguments:

value

The integer to store.

Description:

Construct a `Variant` containing an `int` (n encoding).

Variant(QByteArray)

Construct a `Variant` *containing a QByteArray (*`b64` *encoding).*

Synopsis:

```
#include <qpps/variant.h>

Variant( const QByteArray &value );
```

Arguments:

value

The `QByteArray` to store.

Description:

Construct a `Variant` containing a `QByteArray` (`b64` encoding).

Variant(QJsonDocument)

Construct a `Variant` *containing a* `QJsonDocument` *(*`json` *encoding).*

Synopsis:

```
#include <qpps/variant.h>

Variant( const QJsonDocument &value );
```

Arguments:

value

The `QJsonDocument` to store.

Description:

Construct a `Variant` containing a `QJsonDocument` (`json` encoding).

Variant(QJsonObject)

Construct a `Variant` *containing a* `QJsonObject` *(*`json` *encoding).*

Synopsis:

```
#include <qpps/variant.h>

Variant( const QJsonObject &value );
```

Arguments:

value

The `QJsonObject` to store.

Description:

Construct a `Variant` containing a `QJsonObject` (`json` encoding).

Variant(QString)

Construct a `Variant` *containing a string but with an empty encoding.*

Synopsis:

```
#include <qpps/variant.h>

Variant( const QString &stringValue );
```

Arguments:

stringValue

The string to store.

Description:

Construct a `Variant` containing a `QString` but with an empty encoding.

Variant(QByteArray,QByteArray)

Construct a `Variant` *containing an arbitrary type of value and encoding.*

Synopsis:

```
#include <qpps/variant.h>

Variant( const QByteArray &value,
         const QByteArray &encoding );
```

Arguments:

value

A `QByteArray` containing the raw value to store.

encoding

A `QByteArray` containing the encoding to use.

Description:

Construct a `Variant` containing an arbitrary type of value and encoding. You must first encode the value in the *value* QByteArray before passing it in along with the desired *encoding* (type).

The *value* and *encoding* must not contain null or `\n` byte values and *encoding* must not contain a colon (`:`).

operator==

Test whether two `Variant` *objects are equal (in terms of validity, encoding, and value).*

Synopsis:

```
#include <qpps/variant.h>

bool operator==( const Variant &other ) const;
```

Arguments:

other

The `Variant` on the right side of the equality operator to compare with the one on the left side.

Description:

Test whether two `Variant` objects are equal (in terms of validity, encoding, and value).

Returns:

Returns *true* if the objects are equal, *false* otherwise.

operator!=

Test whether two `Variant` *objects are not equal (in terms of validity, encoding, or value).*

Synopsis:

```
#include <qpps/variant.h>

bool operator!=( const Variant &other ) const;
```

Arguments:

other

The `Variant` on the right side of the inequality operator to compare with the one on the left side.

Description:

Test whether two `Variant` objects are not equal (in terms of validity, encoding, or value).

Returns:

Returns *true* if the objects are *not* equal, *false* otherwise.

Chapter 5 QPlayer API

The `QPlayer` library provides a Qt5 API for sending media commands to the `mm-player` service and for reading media file and playback information from that service. The library replaces the C API of `mm-player` with an object-oriented interface that defines signals for handling media-related events and slots for performing playback operations.

The `QPlayer` API consists of four sections:

- Error and event type enumerations
- Media command classes
- Media information data types
- The `QPlayer` class

All public enumerations, data types, and classes in the QPlayer API are defined within the `QPlayer` namespace. If your app uses other enumerations, data types, or classes with names matching any QPlayer API components, your code must explicitly refer to the QPlayer components by listing the namespace in front of them (e.g., `QPlayer::Metadata`).

Error and event type enumerations

The enumerations in `types.h` define error code constants as well as media source and tracksession event types.

To make your Qt app code more readable, you can use the error code constants when inspecting function return values. The `MediaSourceEventType` and `TrackSessionEventType` enumerations define constants that are encoded in some of the signals emitted by `QPlayer` objects. Qt slots connected to these signals can then check for specific event type constants, which further improves code readability.

Error codes enum

Error codes.

Synopsis:

```
#include <qplayer/types.h>

enum {
    ERROR = -1,
    NO_ERROR = 0
};
```

Data:

ERROR

There was an error with the operation.

NO_ERROR

The operation completed without error.

Library:

`libqplayer`

Description:

Constants describing the possible error codes returned by a `QPlayer` API function.

MediaSourceEventType

Media source event types.

Synopsis:

```
#include <qplayer/types.h>
```

```
typedef enum
{
    MEDIA_ADDED,
    MEDIA_REMOVED,
    MEDIA_UPDATED
} MediaSourceEventType;
```

Data:

MEDIA_ADDED

The media source was added.

MEDIA_REMOVED

The media source was removed.

MEDIA_UPDATED

The media source was updated.

Library:

`libqplayer`

Description:

Media source event types.

TrackSessionEventType

Tracksession event types.

Synopsis:

```
#include <qplayer/types.h>

typedef enum {
    TRACK_SESSION_CREATED,
    TRACK_SESSION_DESTROYED,
    TRACK_SESSION_APPENDED
} TrackSessionEventType;
```

Data:

TRACK_SESSION_CREATED

The tracksession was created.

TRACK_SESSION_DESTROYED

The tracksession was destroyed.

TRACK_SESSION_APPENDED

The tracksession was appended to.

Library:

`libqplayer`

Description:

Tracksession event types.

Media command classes

The `QPlayer` API exposes a set of *command classes* for specifying parameters of media operations and for reading operation results.

When issuing requests to browse media or to read metadata, information on tracks, or playback state information, clients must pass in a pointer to a command object containing the request parameters. The API defines a command class for each operation type. For example, to browse the contents of a folder on a media source, a client must create a `BrowseCommand` object, specify in that object the IDs of the media source and folder that it wants to browse, and then pass in a pointer to that object when calling *QPlayer::browse()*.

For all operations, clients must create a command object that specifies the operation parameters, wait for it to emit a *complete* signal (which indicates the operation succeeded), and then read the operation results by calling the object's *result()* method.

You must wait for the *complete* signal before retrieving the results; otherwise, the results will be empty. The library calls a command object's *setResult()* method to write the results data in the object, but this method is meant for internal use only.

If a command fails, the object instead emits an *error* signal. Your client code can call the *errorMessage()* function to get information about the error.

Command objects include pointers to themselves in their emitted *complete* and *error* signals. These objects delete themselves after the callbacks that process these signals finish executing. So, the client doesn't have to do any memory management for these objects.

Source code sample

The following code sample searches a media source. It's adapted from the reference HMI SearchModel implementation.

```
// Create a pointer to a SearchCommand instance,
// specifying the media source ID and search term parameters
QPlayer::SearchCommand *command =
        new QPlayer::SearchCommand( 1, QStringLiteral("time") );

// Connect a slot to the command's 'complete' signal.
// This slot is called when the command completes successfully.
connect( command, &QPlayer::SearchCommand::complete, this,
         &SearchModel::Private::onSearchResult );

// Connect a slot to the command's 'error' signal.
// This slot is called when the command fails.
connect( command, &QPlayer::SearchCommand::error, this,
         &SearchModel::Private::onSearchError );
```

```
// Start the search by passing the SearchCommand pointer to the
// search() method
q->m_qPlayer->search(command);
```

The complete handler looks like this:

```
void SearchModel::Private::onSearchResult(
                        QPlayer::SearchCommand *command )
{
    // The result of a search command is a list of type
    // QList<QPlayer::MediaNode>. Check if we received a single
    // search result node; if so, output its ID.
    if( command->result().length() == 1 ) {
        qDebug() << command->result().at(0).id;
    }

    // We don't need to clean up the memory allocated for the
    // command because it deletes itself after the 'complete' and
    // 'error' signal handlers finish executing.
    // For this reason, it's important not to keep copies of the
    // command pointer after the signal handlers are called.
    // If the command results are needed outside the handlers,
    // the results should be copied.
}
```

BaseCommand

Defines common features for media commands, such as error message storage and retrieval.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class BaseCommand : public QObject
    {
        Q_OBJECT

    public:

        explicit BaseCommand();

        ~BaseCommand();

        void setErrorMessage( QString message );

        QString errorMessage() const;

    signals:

        void complete( BaseCommand *command );
        // Must be overridden in derived class

        void error( BaseCommand *command );
        // Must be overridden in derived class

    };

}
```

Library:

`libqplayer`

Description:

Defines common features for media commands, such as error message storage and retrieval. All other command classes are derived from this base class and must override the *complete* and *error* signals. The `BaseCommand` class is an abstract class and shouldn't be used directly.

BrowseCommand

Stores browse parameters and results of browse operations.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class BrowseCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit BrowseCommand( int mediaSourceId,
                                QString mediaNodeId,
                                int limit = -1,
                                int offset = 0 );

        inline QList< MediaNode > result()
        {
            return m_result;
        }

        inline void setResult( QList< MediaNode > result )
        {
            m_result = result;
        }

        int mediaSourceId() const;

        QString mediaNodeId() const;

        int limit() const;

        int offset() const;

    signals:

        void complete( BrowseCommand *command );

        void error( BrowseCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores browse parameters and results of browse operations. The results are represented as a list of media nodes. When creating a `BrowseCommand` object, you must provide the IDs of the folder media node to browse and the media source that the node is located on. To start browsing a new media source with an unknown directory structure, pass in `"/"` for the media node ID to indicate the root folder. You can also provide a limit on how many nodes can be stored in the results and specify the offset to start browsing from in the folder.

We recommend using these last two parameters whenever possible because retrieving all media nodes can be very slow due to either a large number of nodes or the device type (e.g., DLNA). You can define the *limit* and *offset* parameters to organize browse results into fixed-size sets of nodes. For example, you can read the first 25 nodes in a media source folder, then the next 25, and so on. This strategy reduces the command processing time and restricts memory usage.

When the underlying player has finished browsing the media source, the `BrowseCommand` object emits a *complete* signal to notify clients that the browse results can now be read. Clients can then call the *result()* method to retrieve the list of media nodes found. From that list, they can read information on individual nodes or further explore the media source by issuing browse requests on nodes found in the list.

If the *browse()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

CreateTrackSessionCommand

Stores tracksession creation parameters and information about the new tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class CreateTrackSessionCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit CreateTrackSessionCommand(
                                    int mediaSourceId,
                                    QString mediaNodeId,
                                    int index = 0,
                                    int limit = -1 );
        inline TrackSession result()
        {
            return m_result;
        }

        inline void setResult( TrackSession result )
        {
            m_result = result;
        }

        int mediaSourceId() const;

        QString mediaNodeId() const;

        int index() const;

        int limit() const;

    signals:

        void complete( CreateTrackSessionCommand *command );

        void error( CreateTrackSessionCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores tracksession creation parameters and information about the new tracksession. A *tracksession* is a sequence of playable tracks. When creating a `CreateTrackSessionCommand` object, you must provide the IDs of the "base" media node, which is used to populate the tracksession, and of the media source that the base node is located on. The exact behavior of tracksession creation depends on the type of the base node and the `mm-player` configuration. For more information, see the *mm_player_create_trksession()* method in the *Multimedia Player Developer's Guide*.

You can also specify the index of the track to be played first and provide a limit on how many items can be stored in the tracksession. These two settings apply only to folders.

When the underlying player has finished creating the tracksession, the `CreateTrackSessionCommand` object emits a *complete* signal to notify clients that the tracksession has been created. Clients can then call the *result()* method to retrieve information about the new tracksession. Note that the tracksession object doesn't contain the actual tracks, but rather an ID that you can pass into the *QPlayer::getTrackSessionItems()* (p. 155) function to retrieve the media nodes of the tracks.

If the *createTrackSession()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

CurrentTrackCommand

Stores information about a track.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class CurrentTrackCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit CurrentTrackCommand();

        inline Track result()
        {
            return m_result;
        }

        inline void setResult( Track result )
        {
            m_result = result;
        }

    signals:

        void complete( CurrentTrackCommand *command );

        void error( CurrentTrackCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores information about a track. When you create a `CurrentTrackCommand` object and pass in its reference in a *getCurrentTrack()* call, the library fills in this object with information on the track currently selected for playback. The object emits a *complete* signal to notify clients when the library finishes writing the track information. Clients can then call the *result()* method to retrieve the information.

If the *getCurrentTrack()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

CurrentTrackPositionCommand

Stores the playback position of the current track.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class CurrentTrackPositionCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit CurrentTrackPositionCommand();

        inline int result()
        {
            return m_result;
        }

        inline void setResult( int result )
        {
            m_result = result;
        }

    signals:

        void complete( CurrentTrackPositionCommand *command );


        void error( CurrentTrackPositionCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores the playback position of the current track. When you create a `CurrentTrackPositionCommand` object and pass in its reference in a *getCurrentTrackPosition()* call, the library writes the playback position of the current track into the object. The object emits a *complete* signal to notify clients when the library finishes writing the playback position. Clients can then call the *result()* method to retrieve the position.

If the *getCurrentTrackPosition()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

ExtendedMetadataCommand

Specifies parameters for retrieving extended metadata and returns extended metadata read.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class ExtendedMetadataCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit ExtendedMetadataCommand(
                                    int mediaSourceId,
                                    QString mediaNodeId,
                                    QStringList properties );

        inline QHash< QString, QVariant > result()
        {
            return m_result;
        }

        inline void setResult(
                const QHash< QString, QVariant > result )
        {
            m_result = result;
        }

        int mediaSourceId() const;

        QString mediaNodeId() const;

        QStringList properties() const;

    signals:

        void complete( ExtendedMetadataCommand *command );

        void error( ExtendedMetadataCommand *command );

    };
}
```

Library:

`libqplayer`

Description:

Specifies parameters for retrieving extended metadata and returns extended metadata read. Here, *extended metadata* refers to nonstandard metadata fields, such as the URL of a media node. When creating an `ExtendedMetadataCommand` object, you must specify the extended metadata fields to read as well as the IDs of the media node being read and of the media source where the node is located.

When the underlying player has finished retrieving the extended metadata from a media node, the object emits a *complete* signal to notify clients that the extended metadata can now be read. Clients can then call the *result()* method to retrieve the extended metadata. Although not essential for helping users locate and play media files, extended metadata provides additional information to assist with browsing and accessing content on media sources.

If the *getExtendedMetadata()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

MediaSourcesCommand

Stores the list of accessible media sources.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class MediaSourcesCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit MediaSourcesCommand();

        inline QList< MediaSource > result()
        {
            return m_result;
        }

        inline void setResult( QList< MediaSource> result )
        {
            m_result = result;
        }

    signals:

        void complete( MediaSourcesCommand *command );

        void error( MediaSourcesCommand *command );

    };

}
```

Library:

```
libqplayer
```

Description:

Stores the list of accessible media sources. The media source information contained in the list includes the names, hardware types, and supported operations for every media source that can be browsed and has content playable by the current player. No parameters are needed when creating a `MediaSourcesCommand` object because this operation is not specific to a media source or media node.

When the underlying player has finished writing the list of media sources, the object emits a *complete* signal to notify clients that information about all accessible media sources can now be read. Clients can then call the *result()* method to retrieve the media source information.

If the *getMediaSources()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

MetadataCommand

Stores metadata retrieval parameters and metadata read from a media node.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class MetadataCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit MetadataCommand( int mediaSourceId,
                                  QString mediaNodeId );

        ~MetadataCommand();

        inline QPlayer::Metadata result()
        {
            return m_result;
        }

        inline void setResult( QPlayer::Metadata result )
        {
            m_result = result;
        }

        int mediaSourceId() const;

        QString mediaNodeId() const;

    signals:

        void complete( MetadataCommand *command );

        void error( MetadataCommand *command );
    };
}
```

Library:

`libqplayer`

Description:

Stores metadata retrieval parameters and metadata read from a media node. Metadata includes a media node's creation details (e.g., artist name, genre, year of release), runtime information (e.g., track length), or display parameters (e.g., width, height). The exact metadata fields defined for a given media node depend on its type—audio, video, or photo. When creating a `MetadataCommand` object, you must specify the IDs of the media node from which you're reading metadata and of the media source on which the node is located.

When the underlying player has finished retrieving the metadata from a media node, the object emits a *complete* signal to notify clients that the metadata can now be read. Clients can then call the *result()* method to retrieve the metadata. Obtaining up-to-date metadata is useful for refreshing the HMI display when the user browses to or starts playing a new media file.

If the *getMetadata()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

PlayerStateCommand

Stores the current player state.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class PlayerStateCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit PlayerStateCommand();

        inline PlayerState result()
        {
            return m_result;
        }

        inline void setResult( PlayerState result )
        {
            m_result = result;
        }
    signals:

        void complete( PlayerStateCommand *command );

        void error( PlayerStateCommand *command );

    };

}
```

Library:

```
libqplayer
```

Description:

Stores the current player state. The state information for a player includes its shuffle mode, repeat mode, playback rate, and playback status (e.g., PLAYING, IDLE). No parameters are needed when creating a `PlayerState` object because this operation isn't specific to a media source or media node.

When the library has finished updating a player's state information, the object emits a *complete* signal to notify clients that the updated player state can now be read. Clients can then call the *result()* method to retrieve the state information.

If the *getPlayerState()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

SearchCommand

Stores search parameters and search results.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class SearchCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit SearchCommand( int mediaSourceId,
                                QString searchTerm,
                                QString filter = "",
                                int limit = -1,
                                int offset = 0 );

        inline QList< MediaNode > result()
        {
            return m_result;
        }

        inline void setResult( QList< MediaNode > result )
        {
            m_result = result;
        }

        int mediaSourceId() const;

        QString searchTerm() const;

        QString filter() const;

        int limit() const;

        int offset() const;

    signals:

        void complete( SearchCommand *command );

        void error( SearchCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores search parameters and search results. The results are returned in a single media node, which you must browse to view individual media nodes found to have metadata matching the search parameters. This design allows clients to create tracksessions containing all the results of a search operation by providing the ID of the media node returned by that operation to a `CreateTrackSessionCommand` object.

When creating a `SearchCommand` object, you must provide a search string and the ID of the media source that you want to search. A media node is added to the results if one of its metadata fields has a value matching the search string. You can specify which fields to examine; by default, the underlying player examines all metadata fields. You can also specify a limit on how many nodes can be stored in the results as well as the offset to start searching from within the root folder of the media source.

When the underlying player has finished searching the media source, the object emits a *complete* signal to notify clients that the search results can now be read. Clients can then call the *result()* method to retrieve the media node containing the search results, which they can browse to read information about individual media nodes matching the search parameters.

If the *search()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

TrackSessionInfoCommand

Stores information about a tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class TrackSessionInfoCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit TrackSessionInfoCommand();

        inline TrackSession result()
        {
            return m_result;
        }

        inline void setResult( TrackSession result )
        {
            m_result = result;
        }

    signals:

        void complete( TrackSessionInfoCommand *command );

        void error( TrackSessionInfoCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores information about a tracksession. When you create a `TrackSessionInfoCommand` object and pass in its reference in a *getCurrentTrackSessionInfo()* call, the library fills in this object with information on the active tracksession for the underlying player.

The object emits a *complete* signal to notify clients when the library finishes writing the tracksession information. Clients can then call the *result()* method to retrieve the information.

If the *getCurrentTrackSessionInfo()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

TrackSessionItemsCommand

Stores parameters for retrieving tracksession items.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    class TrackSessionItemsCommand : public BaseCommand
    {
        Q_OBJECT

    public:

        explicit TrackSessionItemsCommand( int trackSessionId,

                                           int limit = -1,
                                           int offset = 0 );

        inline QList< MediaNode > result()
        {
            return m_result;
        }

        inline void setResult( QList< MediaNode > result )
        {
            m_result = result;
        }

        uint64_t trackSessionId() const;

        int limit() const;

        int offset() const;

    signals:

        void complete( TrackSessionItemsCommand *command );

        void error( TrackSessionItemsCommand *command );

    };

}
```

Library:

`libqplayer`

Description:

Stores parameters for retrieving tracksession items. These items are stored in a list of media nodes, each of which represents an element found in a media source. When creating a `TrackSessionItemsCommand`, you must define the *trackSessionId* parameter to identify the tracksession that you're reading. You may also define the *limit* and *offset* parameters to restrict the number of items retrieved and to specify the tracksession offset to starting reading tracks from.

The object emits a *complete* signal to notify clients when the library finishes filling in the list of media nodes for the tracksession items. Clients can then call the *result()* method to retrieve that list.

If the *getTrackSessionItems()* command fails, the object emits an *error* signal. You can then call *errorMessage()* to retrieve a `QString` describing the error.

Media information data types

The data types defined in `types.h` store information about accessible media files, the active tracksession, the currently selected track, supported metadata fields, and playback state. These data types are used in `QPlayer` function calls to specify media commands and store their outcomes.

The `PlayerState` class defines fields for a player's shuffle and repeat settings as well as its playback status (e.g., `PLAYING`, `STOPPED`) and current playback rate (speed). When one of these playback settings changes for an active player, the `QPlayer` API emits a signal containing a `PlayerState` object that stores the player's latest state.

The `QPlayer` API also allows you to retrieve a list of accessible media sources. The `MediaSource` objects in this list each contain the unique ID, name, hardware type, and other information describing a particular media source. The API function that searches a media source for playable content returns a list of `MediaNode` objects, each of which stores the properties of an element found within the media source. These properties include the media node's type, its unique ID, its URL, and more.

A `TrackSession` object stores only its tracksession ID and length (i.e., number of tracks). Each `Track` object references its related `TrackSession`, `MediaNode`, and `Metadata` object. This last object stores the track's metadata fields, such as the artist name, album title, year of release, the URL of a coverart file, and video dimensions.

MediaNode

Stores properties describing an element found within a media source.

Synopsis:

```
#include <qplayer/types.h>

class MediaNode {

public:
    enum Type {
        UNKNOWN = 0,
        FOLDER,
        AUDIO,
        VIDEO,
        RESERVED1,
        PHOTO,
        NUMBER
    };

    QString id;
    int mediaSourceId;
    QString name;
    QUrl url;
    Type type;
    int count;
};
```

Data:

QString id

Unique ID of the media node.

int mediaSourceId

ID of the media source on which the media node is located.

QString name

Name of the media node.

QUrl url

Media node URL.

Type type

Media node type. Can be one of:

`UNKNOWN`

Unknown file category.

`FOLDER`

Folder.

`AUDIO`

Audio file.

`VIDEO`

Video file.

`RESERVED1`

Reserved for future use.

`PHOTO`

Photo file.

`NUMBER`

End-of-list identifier.

int count

Number of children contained in this node (for folders, `-1` means unknown).

Library:

`libqplayer`

Description:

Stores properties describing an element found within a media source. A media node can be a folder, audio track, video item, or photo. Folders can contain other media nodes (i.e., children). All types of media nodes are found by browsing or searching a media source.

MediaSource

Stores properties describing a connected media source.

Synopsis:

```
#include <qplayer/types.h>

class MediaSource {

public:
    enum Type {
        HDD,
        USB,
        IPOD,
        DLNA,
        BLUETOOTH,
        MTP,
        UNKNOWN
    };

    enum Status {
        NOT_READY,
        READY,
        FIRST_PASS,
        SECOND_PASS,
        THIRD_PASS
    };

    enum Capability {
        PLAY =              (0x00000001),
        PAUSE =             (0x00000002),
        NEXT =              (0x00000004),
        PREVIOUS =          (0x00000008),
        SEEK =              (0x00000010),
        SET_PLAYBACK_RATE = (0x00000020),
        SHUFFLE =           (0x00000040),
        REPEAT_ALL =        (0x00000080),
        REPEAT_ONE =        (0x00000100),
        REPEAT_NONE =       (0x00000200),
        STOP =              (0x00000400),
        JUMP =              (0x00000800),
        GET_POSITION =      (0x00001000),
        METADATA =          (0x00010000),
        SEARCH =            (0x00020000),
        BROWSE =            (0x00040000),
        EXTENDED_METADATA = (0x00080000)
    };

    int id;
    QString uid;
    QString name;
    QString viewName;
    Type type;
    Status status;
    uint64_t capabilities;
};
```

Data:

int id

Unique ID of the media source.

QString uid

Unique ID of the hardware device.

QString name

Media source name.

QString viewName

Name of the view configured for the media source. The view can be changed in the `mm-player` configuration to suit the HMI's needs.

Type type

Hardware type. Can be one of:

`HDD`

Local drive.

`USB`

USB storage device.

`IPOD`

iPod.

`DLNA`

DLNA device.

`BLUETOOTH`

Bluetooth device.

MTP

Device with MTP files (e.g., Android, Win7/8 phone).

UNKNOWN

Customized media source.

Status status

Media source status. Can be one of:

NOT_READY

The media source isn't ready because the device is connected but hasn't been synchronized.

READY

The media source is ready, meaning it's connected and synchronized and its status can be read.

FIRST_PASS

The file information from the media source has been synchronized.

SECOND_PASS

The media metadata from the media source has been synchronized.

THIRD_PASS

The playlist entry information for the media source has been synchronized.

uint64_t capabilities

A flag field indicating the supported browsing and playback operations. Supported flags include:

PLAY

Playback is supported.

PAUSE

Playback can be paused.

NEXT

You can skip to the next track.

PREVIOUS

You can skip to the previous track.

SEEK

You can seek to a specific playback position.

SET_PLAYBACK_RATE

Playback speed can be adjusted.

SHUFFLE

Playback can be shuffled (i.e., randomized)

REPEAT_ALL

You can repeat all tracks in the same order.

REPEAT_ONE

You can repeat one track continuously.

REPEAT_NONE

You can disable repeating.

STOP

Playback can be stopped.

JUMP

You can jump to another track within the active tracksession.

GET_POSITION

You can retrieve the current playback position.

METADATA

You can retrieve metadata from media nodes.

SEARCH

You can retrieve media nodes with metadata properties matching a search string.

BROWSE

You can browse a media node within a media source.

EXTENDED_METADATA

You can retrieve extended metadata (i.e., nonstandard properties) from media nodes.

Library:

`libqplayer`

Description:

Stores properties describing a connected media source. This class defines enumerations that specify the possible values for a media source's hardware type and connection status as well as the flags that represent various media operations. When examining the fields that store these settings in a `MediaSource` object, your code can compare the field values to specific enumeration constants, making it more readable.

Metadata

Stores metadata fields containing creation and playback information for a media node.

Synopsis:

```
#include <qplayer/types.h>

class Metadata {

public:
    QString title;
    int duration;
    QUrl artwork;
    QString artist;
    QString album;
    QString genre;
    QString year;
    int width;
    int height;
    int disc;
    int track;
    QString reserved;
};
```

Data:

QString title

Media file title.

int duration

Track duration (in milliseconds).

QUrl artwork

URL of filepath for artwork (NULL if there's no artwork).

QString artist

Artist name.

QString album

Album name.

QString genre

Genre.

QString year

Year of creation.

int width

Width (in pixels).

int height

Height (in pixels).

int disc

Disc number (`-1` if not applicable).

int track

Track index (`-1` if not applicable).

QString reserved

Reserved for future use.

Library:

`libqplayer`

Description:

Stores metadata fields containing creation and playback information for a media node.

PlayerState

Stores playback status properties.

Synopsis:

```
#include <qplayer/types.h>

class PlayerState {

public:
    enum PlayerStatus {
        DESTROYED,
        IDLE,
        PLAYING,
        PAUSED,
        STOPPED
    };

    enum RepeatMode {
        QP_REPEAT_OFF,
        QP_REPEAT_ALL,
        QP_REPEAT_ONE
    };

    enum ShuffleMode {
        QP_SHUFFLE_OFF,
        QP_SHUFFLE_ON
    };

    ShuffleMode shuffle;
    RepeatMode repeat;
    PlayerStatus status;
    float rate;
} ;
```

Data:

ShuffleMode shuffle

Shuffle mode. Can be one of:

`SHUFFLE_OFF`

Shuffling is off; tracks will be played sequentially.

`SHUFFLE_ON`

Shuffling is on; tracks will be played in a random order.

RepeatMode repeat

Repeat mode. Can be one of:

`REPEAT_OFF`

No tracks will be repeated (playback will stop when the end of the active tracksession is reached).

`REPEAT_ALL`

All tracks will be repeated in the same order (playback will loop).

`REPEAT_ONE`

The current track will be continuously repeated.

PlayerStatus status

The player's status, which reflects its current playback support and activity. Can be one of the following:

`STATUS_DESTROYED`

Reserved for future use.

`STATUS_IDLE`

The player is created but no tracksession is defined so playback is currently not possible.

`STATUS_PLAYING`

A track is currently playing.

`STATUS_PAUSED`

Playback is paused.

`STATUS_STOPPED`

Playback is stopped, no track is selected, or an error has occurred.

float rate

Playback rate (i.e., the speed of playback).

Library:

`libqplayer`

Description:

Stores playback status properties. This class defines enumerations that specify the possible settings for a player's status and its repeat and shuffle modes. When examining the fields that store these settings in a `PlayerState` object, your code can compare the field values to specific enumeration constants, making it more readable.

Track

Stores properties describing the currently selected track within a tracksession.

Synopsis:

```
#include <qplayer/types.h>

class Track {

public:
    int index;
    uint64_t tsid;
    MediaNode mediaNode;
    Metadata metadata;
};
```

Data:

int index

Position of the track within the tracksession.

uint64_t tsid

ID of the associated tracksession.

MediaNode mediaNode

Media node on which this track is based.

Metadata metadata

Track metadata.

Library:

`libqplayer`

Description:

Stores properties describing the currently selected track within a tracksession.

TrackSession

Stores the ID and length of a tracksession.

Synopsis:

```
#include <qplayer/types.h>

class TrackSession {

public:
    uint64_t id;
    int length;
};
```

Data:

uint64_t id

Tracksession ID.

int length

Number of tracks within the tracksession.

Library:

`libqplayer`

Description:

Stores the ID and length of a tracksession.

QPlayer class

Sends media commands to `mm-player` and emits signals for media state changes.

Synopsis:

```
#include <qplayer/qplayer.h>

namespace QPlayer {

    static const QString ROOT_MEDIA_NODE_ID =
                    QStringLiteral("/");

    class QPLAYER_EXPORT QPlayer : public QObject
    {
        Q_OBJECT

    public:

        explicit QPlayer( const QString playerName,
                          QObject *parent = 0 );

        ~QPlayer();

        void getMediaSources( MediaSourcesCommand *command );

        void getPlayerState( PlayerStateCommand *command );

        void getCurrentTrack( CurrentTrackCommand *command );

        void getCurrentTrackPosition(
                     CurrentTrackPositionCommand *command );

        void browse( BrowseCommand *command );

        void search( SearchCommand *command );

        void getMetadata( MetadataCommand *command );

        void getExtendedMetadata(
                       ExtendedMetadataCommand *command );

        void createTrackSession(
                       CreateTrackSessionCommand *command );

        int destroyTrackSession( uint64_t tsid );

        void getTrackSessionItems(
                       TrackSessionItemsCommand *command );

        void getCurrentTrackSessionInfo(
                           TrackSession *trackSession );

    public Q_SLOTS:

        void play();

        void pause();
        void stop();

        void next();

        void previous();

        void seek( const int position );

        void jump( const int index );

        void setPlaybackRate( const float rate );

        void setShuffleMode( const PlayerState::ShuffleMode
mode );
        void setRepeatMode( const PlayerState::RepeatMode mode
 );
    Q_SIGNALS:

        void playerReady();

        void mediaSourceChanged(
                            const MediaSourceEventType type,
                            const MediaSource &mediaSource );

        void trackChanged( Track track );

        void trackPositionChanged( int trackPosition );

        void trackSessionChanged( TrackSessionEventType type,

                                    TrackSession trackSession );


        void playerStateChanged( PlayerState playerState );

    };

}
```

Library:

`libqplayer`

Description:

Sends media commands to `mm-player` and emits signals for media state changes. The `QPlayer` class is the main class that your apps use to interact with the `QPlayer` library.

When creating `QPlayer` objects, you must name the player you want to connect with. The reference HMI uses the same player in the Media Player app (which provides visual media controls) and the ASR subsystem (which supports voice commands related to media). For more information about players, see the *mm_player_open()* function in the *Multimedia Player Developer's Guide*.

The new `QPlayer` object waits (if necessary) until `mm-player` is ready before trying to open the specified player. If successful, it emits the *playerReady* signal. You can then use the object to browse media sources, retrieve metadata, create tracksessions, and manage playback. The object will emit signals when the player's state changes. If it can't open the player, the `QPlayer` library logs an error.

Results-based media commands, such as media node searches or metadata retrieval requests, use *command classes* (p. 119). This means you must specify operation details (e.g., which folder to search) in a command object, which the library also uses to return the results. For commands that retrieve state information (e.g., player status or information on the current track), the command object is used only for returning results. Note that you must wait for any command object to emit a *complete* signal before retrieving the operation results; otherwise, the results will be empty.

Playback functions don't require command objects because they carry out relatively basic actions, such as starting and stopping playback or changing tracks. All playback functions are implemented as Qt slots, so you can connect them to signals so that they run in response to user actions. For example, you can connect playback functions to signals emitted by a GUI after the user taps an HMI button.

This class also defines signals that, when emitted, indicate changes to:

- media source connections
- the active tracksession
- the track selected for playback
- the playback position
- the state of the underlying player

Public constants used by `QPlayer`

Constants used by the `QPlayer` class to specify special folders for browsing.

ROOT_MEDIA_NODE_ID

The root media node ID for all media sources.

Synopsis:

```
#include <qplayer/qplayer.h>

static const QString ROOT_MEDIA_NODE_ID = QStringLiteral("/");
```

Description:

The root media node ID for all media sources.

Public functions in `QPlayer`

Functions defined in the `QPlayer` class for exploring media sources, reading metadata, defining tracksessions, and controlling playback.

browse()

Browse a media source for media nodes.

Synopsis:

```
#include <qplayer/qplayer.h>

void browse( BrowseCommand *command );
```

Arguments:

command

A pointer to a `BrowseCommand` object that specifies the folder to browse and other operation parameters. This object will also hold the operation results (i.e., the media nodes found while browsing).

Description:

Browse a media source for media nodes. The underlying player browses the folder media node indicated by the *BrowseCommand* (p. 121) object.

The media nodes found during browsing are stored in this same object. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve the media node information. Note that media nodes in the results can be folders (which can contain other media nodes) or individual media files such as audio tracks, videos, or photos.

createTrackSession()

Create a tracksession from a media node.

Synopsis:

```
#include <qplayer/qplayer.h>

void createTrackSession( CreateTrackSessionCommand *command );
```

Arguments:

command

A pointer to a `CreateTrackSessionCommand` object that specifies operation parameters such as the media node for creating the tracksession. The `TrackSession` object created to represent the new tracksession is then written to the object referenced in this argument.

Description:

Create a tracksession from a media node. A *tracksession* is a sequence of tracks with a particular playback order. The new tracksession is filled with the tracks (i.e., media nodes) found within the path of the media node specified in the *CreateTrackSessionCommand* (p. 122) object.

Information on the newly created tracksession is stored in this same object. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve that information.

destroyTrackSession()

Destroy a tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

int destroyTrackSession( uint64_t tsid );
```

Arguments:

tsid

The ID of the tracksession to destroy.

Description:

Destroy a tracksession.

Returns:

Returns `0` when the tracksession was destroyed successfully, `-1` if there was an error.

getCurrentTrack()

Get information about the track currently selected for playback.

Synopsis:

```
#include <qplayer/qplayer.h>

void getCurrentTrack( CurrentTrackCommand *command );
```

Arguments:

command

A pointer to a `CurrentTrackCommand` object, which will hold information on the current track.

Description:

Get information about the track currently selected for playback. The library fills in the *CurrentTrackCommand* (p. 124) object referenced in *command* with information describing the current track. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve that information.

getCurrentTrackPosition()

Get the playback position of the current track.

Synopsis:

```
#include <qplayer/qplayer.h>

void getCurrentTrackPosition(
                    CurrentTrackPositionCommand *command );
```

Arguments:

command

A pointer to a `CurrentTrackPositionCommand` object, which will hold the current playback position.

Description:

Get the playback position of the current track. The library fills in the *CurrentTrackPositionCommand* (p. 125) object referenced in *command* with the current playback position. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve that position.

getCurrentTrackSessionInfo()

Get information about the active tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

void getCurrentTrackSessionInfo(
                    TrackSessionInfoCommand *command );
```

Arguments:

command

A pointer to a `TrackSessionInfoCommand` object, which will hold information on the active tracksession.

Description:

Get information about the active tracksession. The library fills in the *TrackSessionInfoCommand* (p. 132) object referenced in *command* with tracksession information. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve that information.

getExtendedMetadata()

Get extended metadata associated with a media node.

Synopsis:

```
#include <qplayer/qplayer.h>

void getExtendedMetadata( ExtendedMetadataCommand *command );
```

Arguments:

command

A pointer to an `ExtendedMetadataCommand` object that specifies the media node being read and which of its fields are being read. The extended metadata values are written to this object.

Description:

Get extended metadata associated with a media node. Here, *extended metadata* refers to nonstandard metadata values, such as the URL of a media node, that aren't returned by *getMetadata()*. The *ExtendedMetadataCommand* (p. 126) object referenced in *command* contains the IDs of the node to read and of the media source where the node is located as well as the fields to read.

The extended metadata values extracted from the media node are stored in this same object. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve those values.

getMediaSources()

Get a list of all connected media sources.

Synopsis:

```
#include <qplayer/qplayer.h>

void getMediaSources( MediaSourcesCommand *command );
```

Arguments:

command

A pointer to a `MediaSourcesCommand` object, which will hold the list of media sources.

Description:

Get a list of all connected media sources. The library fills in the *MediaSourcesCommand* (p. 127) object referenced in *command* with information describing each connected media source. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve that information.

getMetadata()

Get metadata associated with a media node.

Synopsis:

```
#include <qplayer/qplayer.h>

void getMetadata( MetadataCommand *command );
```

Arguments:

command

A pointer to a `MetadataCommand` object that specifies the media node being read. The node's metadata values are then written to the same object.

Description:

Get metadata associated with a media node. The *MetadataCommand* (p. 128) object referenced in *command* contains the node to read metadata from and the media source where the node is located.

This same object stores the metadata values extracted from the media node. When your client receives the *complete* signal, it can call its *result()* function to retrieve those values. Note that the metadata read depends on the type of the media node. For example, an audio track has an artist name and genre but not a width or height. Video and photo files, however, do have those last two fields.

getPlayerState()

Get the current player state.

Synopsis:

```
#include <qplayer/qplayer.h>

void getPlayerState( PlayerStateCommand *command );
```

Arguments:

command

A pointer to a `PlayerStateCommand` object, which will hold information on the player state.

Description:

Get the current player state. The library fills in the *PlayerStateCommand* (p. 129) object referenced in *command* with information describing the player's state. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve that information.

getTrackSessionItems()

Get the media nodes of the items in a tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

void getTrackSessionItems(
                    TrackSessionItemsCommand *command );
```

Arguments:

command

A pointer to a `TrackSessionItemsCommand` object that specifies the tracksession being read. A list of media nodes corresponding to the tracksession's items is then written to this object.

Description:

Get the media nodes of the items in a tracksession. The ID of the tracksession being read is specified in the *TrackSessionItemsCommand* (p. 133) object.

The media nodes corresponding to the tracksession items are stored in this same object. When your client receives the *complete* signal, it can call the *result()* function on this object to retrieve those media nodes.

QPlayer()

Create a `QPlayer` instance, opening the specified player.

Synopsis:

```
#include <qplayer/qplayer.h>

explicit QPlayer( const QString playerName,
                  QObject *parent = 0 );
```

Arguments:

playerName

The name of the player to open.

parent

A pointer to the parent `QObject`. This parameter links a `QPlayer` to a `QObject` (or a subtype) so the new child object gets deleted when its parent is deleted. By default, no parent is assigned.

Description:

Create a `QPlayer` instance, opening the player specified in *playerName*. This player is managed by the `mm-player` service and carries out the media requests sent by the client through the `QPlayer` object.

~QPlayer()

Destroy the QPlayer object.

Synopsis:

```
#include <qplayer/qplayer.h>

~QPlayer();
```

Description:

Destroy the QPlayer object.

search()

Search a media source for media nodes that have metadata matching a search string.

Synopsis:

```
#include <qplayer/qplayer.h>

void search( SearchCommand *command );
```

Arguments:

command

A pointer to a `SearchCommand` object that specifies the media source to search, the search string for filtering the results, and other operation parameters. This object will return the operation results (i.e., the results media node) when requested.

Description:

Search a media source for media nodes that have metadata matching a search string. The media source to search, the search string, and the metadata fields to compare against the search string are specified in the *SearchCommand* (p. 130) object referenced in *command*.

The search operation returns a single results node, which you must then browse to view the individual media nodes in the results. When your client receives the *complete* signal, it can call the *result()* function on the command object to retrieve the results node. Note that the search results may include all types of media files—audio tracks, videos, and photos.

Public slots in `QPlayer`

Slots defined in the `QPlayer` class for controlling playback.

jump()

Jump to another track in the active tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

void jump( const int index );
```

Arguments:

index

The index of the newly selected track in the active tracksession.

Description:

Jump to another track in the active tracksession. This function changes the “current” track (i.e., the tracksession item selected for playback) to the track at position *index*. If playback is active when this function is called, the player starts playing the track immediately. If playback isn't active, the track will be played when playback resumes.

next()

Skip to the next track in the tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

void next();
```

Description:

Skip to the next track in the tracksession. The track considered the “next track” depends on the *shuffle* (p. 144) and *repeat* (p. 144) settings.

When the repeat mode is REPEAT_ONE or REPEAT_OFF, the next track is the track immediately following the current track in either the sequential playback list (if shuffling is off) or in the randomized list (if shuffling is on). If the current track is the last track in the list, playback stops (because there's no next track).

When the repeat mode is REPEAT_ALL, the function behaves similarly except that if the current track is the last track, the next track is the first track in the list (because playback is looped).

pause()

Pause playback.

Synopsis:

```
#include <qplayer/qplayer.h>

void pause();
```

Description:

Pause playback. Calling this function changes the player *status* (p. 144) to STATUS_PAUSED but maintains the current playback position. This way, you can resume playback at the exact position where you paused it.

play()

Begin or resume playback.

Synopsis:

```
#include <qplayer/qplayer.h>

void play();
```

Description:

Begin or resume playback. Calling this function changes the player *status* (p. 144) to STATUS_PLAYING. This status setting remains in effect until either you pause playback or the end of the tracksession is reached and repeating is disabled.

The track that begins playing is the one selected as the "current" track in the active tracksession. When this track finishes playing, the player chooses a new track to play based on its *shuffle* (p. 144) and *repeat* (p. 144) settings. To see which track is currently selected, call *getCurrentTrack()* (p. 151). To see the current playback position, call *getCurrentTrackPosition()* (p. 152).

previous()

Skip to the previous track in the tracksession.

Synopsis:

```
#include <qplayer/qplayer.h>

void previous();
```

Description:

Skip to the previous track in the tracksession. The track considered the "previous track" depends on the *shuffle* (p. 144) and *repeat* (p. 144) settings.

When the repeat mode is REPEAT_ONE or REPEAT_OFF, the previous track is the track immediately preceding the current track in either the sequential playback list (if shuffling is off) or in the randomized list (if shuffling is on). If the current track is the first track in the list, playback stops (because there's no previous track).

When the repeat mode is REPEAT_ALL, the function behaves similarly except that if the current track is the first track, the previous track is the last track in the list (because playback is looped).

seek()

Seek to a position within the current track.

Synopsis:

```
#include <qplayer/qplayer.h>

void seek( const int position );
```

Arguments:

position

The new track position (in milliseconds from the start of the track).

Description:

Seek to a position within the current track. The value in *position* is the number of milliseconds from the start of the track (e.g., `2500`).

setPlaybackRate()

Set the playback rate.

Synopsis:

```
#include <qplayer/qplayer.h>

void setPlaybackRate( const float rate );
```

Arguments:

rate

The new playback rate, relative to a normal rate of `1.0`.

Description:

Set the playback rate (speed). The floating-point value in *rate* is relative to a normal rate of `1.0`. A value of `0` pauses playback. Negative numbers cause the media to play in reverse.

setRepeatMode()

Set the player's repeat mode.

Synopsis:

```
#include <qplayer/qplayer.h>

void setRepeatMode( const PlayerState::RepeatMode mode );
```

Arguments:

mode

The new repeat mode.

Description:

Set the player's repeat mode. The *repeat mode* (p. 144) allows you to repeatedly play an individual track or a sequence of tracks.

The `REPEAT_ONE` mode causes the player associated with the `QPlayer` object to play the same track continuously until you either stop playback or skip to another track.

The `REPEAT_ALL` mode makes the player play all the tracks in the active tracksession and then loop back to the beginning of the tracksession. The playback order is either sequential (when shuffling is off) or random (when shuffling is on).

If the repeat mode is `REPEAT_OFF`, the player plays all the tracks exactly once but stops when it reaches the end of the tracksession. By default, the repeat mode is `REPEAT_OFF`.

setShuffleMode()

Set the player's shuffle mode.

Synopsis:

```
#include <qplayer/qplayer.h>

void setShuffleMode( const PlayerState::ShuffleMode mode );
```

Arguments:

mode

The new shuffle mode.

Description:

Set the player's shuffle mode. The *shuffle mode* (p. 144) determines which of two lists the player associated with the `QPlayer` object uses to select the next track for playback.

When the mode is `SHUFFLE_ON`, the player uses a randomized track list, which indexes tracks in an order different from their order in the media source. For example, when the track listed as number 2 on its album finishes playing, the next track played could be any other track on the album (including the track listed as number 3).

When the mode is `SHUFFLE_OFF`, the player uses the sequential track list, which reflects the track order in the media source. In this case, when track number 2 finishes playing, track number 3 will play next.

When you set the mode to `SHUFFLE_ON`, the player generates a new randomized playback list. So you can keep randomized playback enabled and just change to a different random order by calling this function multiple times with this setting. By default, the shuffle mode is set to `SHUFFLE_OFF`.

stop()

Stop playback.

Synopsis:

```
#include <qplayer/qplayer.h>

void stop();
```

Description:

Stop playback. Calling this function changes the player *status* (p. 144) to `STATUS_STOPPED` and resets the playback position. When you resume playback, the currently selected track will begin playing from the start again. While playback is stopped, you can change the current track if you want to play something different when playback resumes.

Signals in `QPlayer`

Signals emitted by `QPlayer` objects for indicating changes to media source connections, the active tracksession, the current track, the playback position, and the player state.

mediaSourceChanged()

Emitted when a media source was added, removed, or modified.

Synopsis:

```
#include <qplayer/qplayer.h>

void mediaSourceChanged( const MediaSourceEventType type,
                         const MediaSource &mediaSource );
```

Arguments:

type

The media source event type.

mediaSource

The media source that the event applies to.

Description:

Emitted when a media source was added, removed, or modified. The event type is stored in *type*, as a constant defined by the *MediaSourceEventType* (p. 116) enumeration. The object in *mediaSource* stores information on the media source that the event applies to.

playerReady()

Emitted when the underlying player has been initialized

Synopsis:

```
#include <qplayer/qplayer.h>

void playerReady();
```

Description:

Emitted when the underlying player has been initialized. Your client must wait for this signal before issuing any commands through the object. Any new `QPlayer` object must wait for the `mm-player` service to be flagged as ready before trying to open a connection to the player. After the object successfully opens the player connection, it emits the *playerReady()* signal.

playerStateChanged()

Emitted when a configuration setting or the playback status has changed for a player.

Synopsis:

```
#include <qplayer/qplayer.h>

void playerStateChanged( PlayerState playerState );
```

Arguments:

playerState

The updated state of the player, which includes its latest shuffle mode, repeat mode, playback rate, and playback status (e.g., `PLAYING`, `IDLE`).

Description:

Emitted when a configuration setting or the playback status has changed for a player. The object in *playerState* stores the latest state information of the affected player.

trackChanged()

Emitted when a new track is selected for playback.

Synopsis:

```
#include <qplayer/qplayer.h>

void trackChanged( Track track );
```

Arguments:

track

The new “current” track, which is either playing now or will be played when playback resumes.

Description:

Emitted when a new track is selected for playback. This happens when the user skips to another tracksession item or when the player moves to another track automatically, either because the previous track finished playing or a playback error occurred. In all these cases, the object in *track* contains information on the newly selected track.

trackPositionChanged()

Emitted when the current track's playback position was changed.

Synopsis:

```
#include <qplayer/qplayer.h>

void trackPositionChanged( int trackPosition );
```

Arguments:

trackPosition

The new playback position (in milliseconds from the start of the track).

Description:

Emitted when the current track's playback position was changed. The new playback position is stored in *trackPosition*.

trackSessionChanged()

Emitted when a tracksession was created, destroyed, or appended to.

Synopsis:

```
#include <qplayer/qplayer.h>

void trackSessionChanged( TrackSessionEventType type,
                          TrackSession trackSession );
```

Arguments:

type

The tracksession event type.

trackSession

The updated information for the tracksession that the event applies to.

Description:

Emitted when a tracksession was created, destroyed, or appended to. The event type is stored in *type*, as a constant defined by the *TrackSessionEventType* (p. 117) enumeration. The object in *trackSession* stores information on the tracksession that the event applies to.

Index