Qt Developer's Guide

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: June 01, 2015

Contents

About This Guide

This document explains how to set up a host system for Qt development and how to perform all tasks in the development lifecycle for Qt apps.

The QNX Apps and Media reference image includes many sample Qt apps, which provide useful programming references for developing apps for various domains (e.g., media playback, camera display, system control). The tutorials in this document show you how to use Qt Creator to define projects, specify a basic UI, build and package apps, and deploy and run them on a target system.

The pre-built Qt distribution available with the QNX SDK for Apps and Media 1.1 is an optimized port of the Qt Community version and has been made available as a convenience for our customers. Although this version of Qt is not a QNX commercially licensed product, you can obtain Qt support from QNX under a Custom Services Plan (CSP). Qt Enterprise and support for Qt Enterprise is available from The Qt Company (*http://www.qt.io*). For more information about Qt licensing, see *http://www.qt.io/licensing/*.

To find out about:	See:
The components needed to develop Qt apps and where to find these components	*QNX Qt Development Tools* (p. 9)
How to access the QNX Browser and deliver HTML5 content from Qt apps	*QNX Browser Invocation from Qt* (p. 11)
How to install and configure the Qt development tools on your host system	*Preparing your host system for Qt development* (p. 13)
How to develop, package, deploy, and run Qt apps on a QNX Apps and Media target	*Creating and running Qt apps* (p. 25)
How to build a library and dynamically link it into Qt apps	*Building libraries for Qt apps* (p. 59)
How to develop and display a Qt HMI on a QNX Apps and Media target	*Writing an HMI* (p. 71)
How to run applications written for QNX Apps and Media 1.0 on QNX Apps and Media 1.1 targets	*Building the HMI application for a QNX target* (p. 78)

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`
Constants	NULL
Data types	**unsigned short**
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.

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

WARNING: 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 typically use a forward slash (/) as a delimiter in 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 Tools

To write Qt apps for QNX devices, you need to install Qt version 5.3.1 and Qt Creator version 3.2.1 onto your development system.

Before you can install the Qt tools, your system must have these platform installations:

- QNX SDP 6.6
- QNX SDK for Apps and Media 1.1

With this platform support, you can configure and start using these Qt development tools:

Qt runtime

Our Qt runtime package is based on Qt version 5.3.1 and contains a version of the build tools (e.g., **qmake**, **qcc**) adapted to generate binary and library files for BlackBerry 10 OS.

Qt Creator

This IDE lets you manage projects for Qt applications, edit C++ and QML source files, and add project resources such as images. This release officially supports version 3.2.1 of Qt Creator, which you must configure to use the build tools in the installed Qt runtime package.

Details on accessing and installing the Qt runtime and IDE are given in *Preparing your host system for Qt development* (p. 13).

Source code for sample Qt apps

The QNX SDK for Apps and Media installers include a zipped folder (**appsmedia_qt_source_v1_1.zip**) containing the source code for many sample Qt apps. These apps provide programming references for implementing functions such as media playback, photo viewing, and displaying an HMI that lists the installed apps.

The installers copy the Qt source code package to the **source** directory within the root directory of the QNX SDP 6.6 installation (e.g., **/usr/qnx660/source/appsmedia_qt_source_v1_1.zip**). You can extract the files containing the Qt source code to any location and examine their contents; however, you can't modify or rebuild the sample apps without installing and configuring the Qt development tools as described in this document.

The sample apps are:

Home screen

An HMI, built from Qt, that displays a status bar and the icons of the installed apps, which users can tap to launch those apps

IP Camera

Displays a video feed supplied by an RTP/IP-based camera

Media Player

Browses and plays audio and video content

Photo Viewer

Displays picture files

QtSimpleHmi

A basic HMI written as a stand-alone application with no packaging

Settings

Provides controls to configure the system

These same Qt apps are part of the shipped images but only their binaries and runtime resources (e.g., icon files) are included in the images. When you extract the Qt source code package on your host system, each sample app listed above is found in a directory with the same name. There's also the **Common** directory, which stores classes useful to many types of apps.

QNX Browser Invocation from Qt

Although Qt includes programming interfaces for accessing web browsers, these components aren't supported by the QNX Qt runtime for this release. Instead, your Qt code can start one of our sample QNX browsers or a custom browser, by using the **launcher** service.

The shipped image includes two browsers:

- BrowserLite, a basic web browser built from HTML5 and Cordova plugins
- Browser, a fully-functional browser with advanced features (e.g., browsing history, URL bookmarking) built from HTML5 and JavaScript

Both browsers are packaged and installed as apps (as opposed to stand-alone applications with their own HMI) and hence, can be launched from Qt code by writing a command to a Persistent Publish/Subscribe (PPS) object monitored by the **launcher** service. An example of using Qt to launch an app is the Home screen app. The source code of this app is found in the Qt source code bundle (**appsmedia_qt_source_v1_1.zip**), which is part of the QNX SDK for Apps and Media installer package. Specifically, you can examine the code in the **Homescreen/app/launcher** subdirectory to see how to format and send the `start` command through PPS.

For more information about the Home screen sample and how users interact with it to launch apps, see the *User's Guide*. For a reference of the PPS control object used by **launcher**, see the **/pps/services/launcher/control** entry in the *PPS Objects Reference*.

If you want to deliver HTML5 content to the user without running a browser, your Qt apps can start any app written with HTML5 and related technologies, using the same mechanism of sending an app launch request to **launcher** through PPS.

Chapter 2
Preparing your host system for Qt development

To write Qt applications, you must install the Qt runtime and Qt Creator IDE and then configure the IDE to use our build tools and to target applications to a QNX device.

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 QNX Apps and Media.

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.
- An installation of QNX SDK for Apps and Media 1.1 on your host system. This latter platform depends on some critical SDP patches that fix key subsystems (e.g., audio, graphics); the list of required patches is given in the platform's *Installation Note*.
- A target system running QNX Apps and Media 1.1 that's connected to the same network as the host system and that has a valid IP address.

Installing QNX QDF and Qt Creator

QNX Qt 5.3.1 Development Framework (QNX QDF) is a package containing the Qt runtime needed for building Qt apps. Qt Creator is the IDE that you use to write, debug, and build the apps. You need to install both components before you can develop Qt apps for QNX Apps and Media systems.

To install the Qt development tools on your host system:

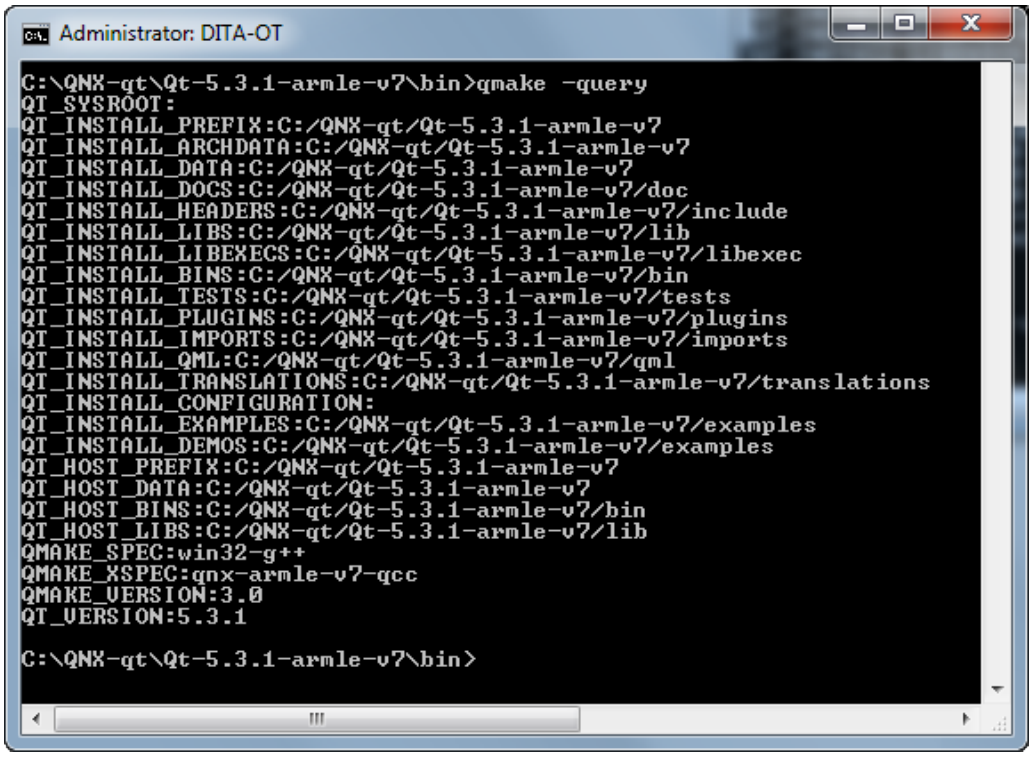
1. Locate the *Installation Note* applicable to your host OS by visiting our website, ***www.qnx.com***, and going to the QNX SDK for Apps and Media 1.1 Download area.

   To find the right documentation for your host, look for headings with names similar to "`QNX Qt 5.3.1 Development Framework (Windows Hosts)`" or "`QNX Qt 5.3.1 Development Framework (Linux Hosts)`", then click `See Installation/Release notes...` to access the supporting documentation.

2. Follow the instructions in the *Installation Note* to access, download, and install QNX QDF and Qt Creator.

   The installation dialog will prompt you for the directory to install Qt into, which we refer to as *QT_BASEDIR*. In this example, we use the default directory of **C:\QNX-qt** on a Windows host.

3. Verify the correct paths of the QNX QDF build resources by opening an OS terminal, navigating to the location of the **qmake** version suitable for your intended QNX target, and typing `qmake -query`:



   The path of the **qmake** utility is ***QT_BASEDIR*\Qt-5.3.1-***variant*\bin**, where *variant* is **x86** or **armle-v7**, depending on your target's processor type. Note that on Linux the directory separators would be forward slashes (`/`).

In this example, we use a Windows host and a target system that has an armle-v7 processor, so we query the properties of **C:\QNX-qt\Qt-5.3.1-armle-v7\bin\qmake.exe**. Regardless of your host OS and target type, the paths of the build resources shown in the output should match the first few directory levels in the **qmake** path.

After QNX QDF and Qt Creator are successfully installed, 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.

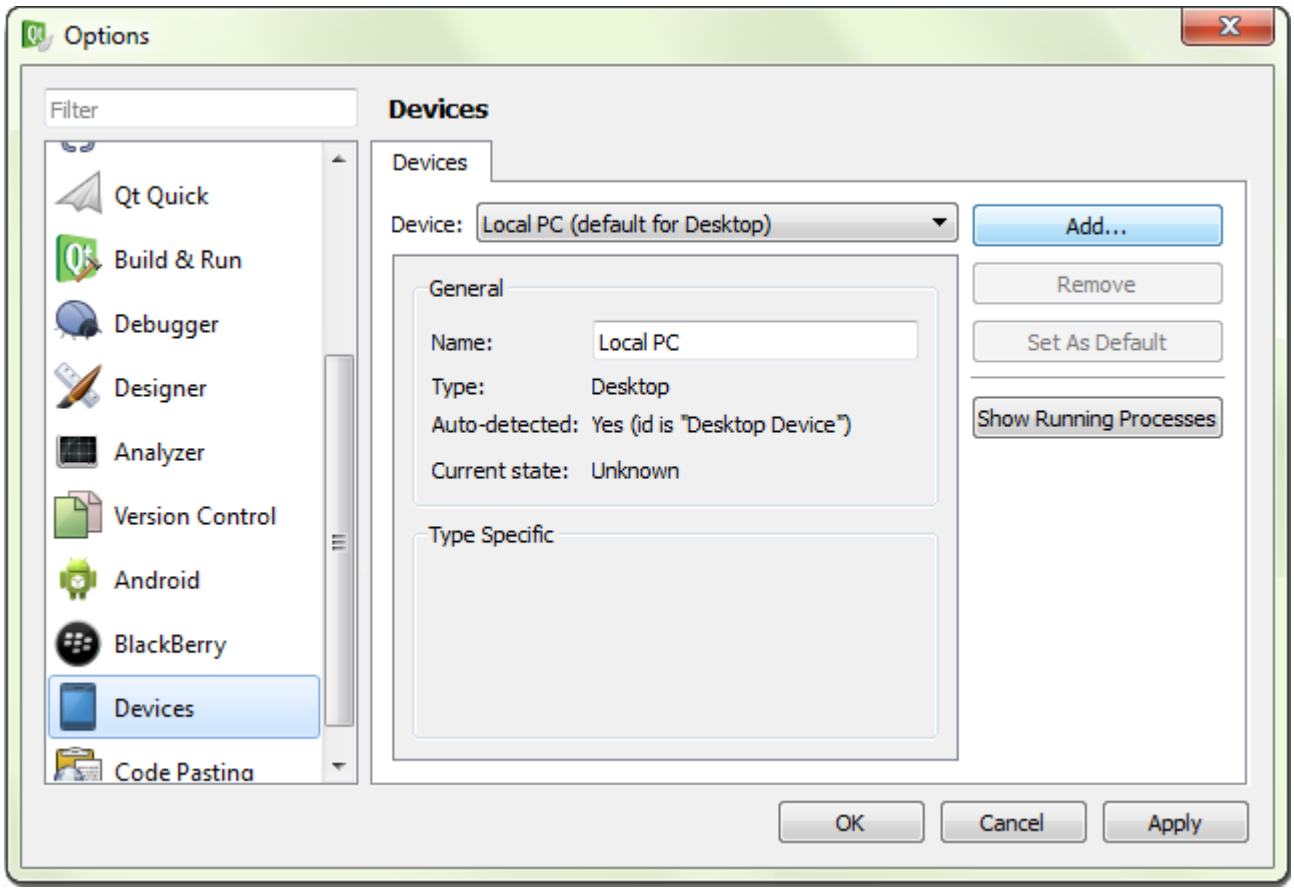
Configuring a QNX device in Qt Creator

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 Apps and Media.

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.

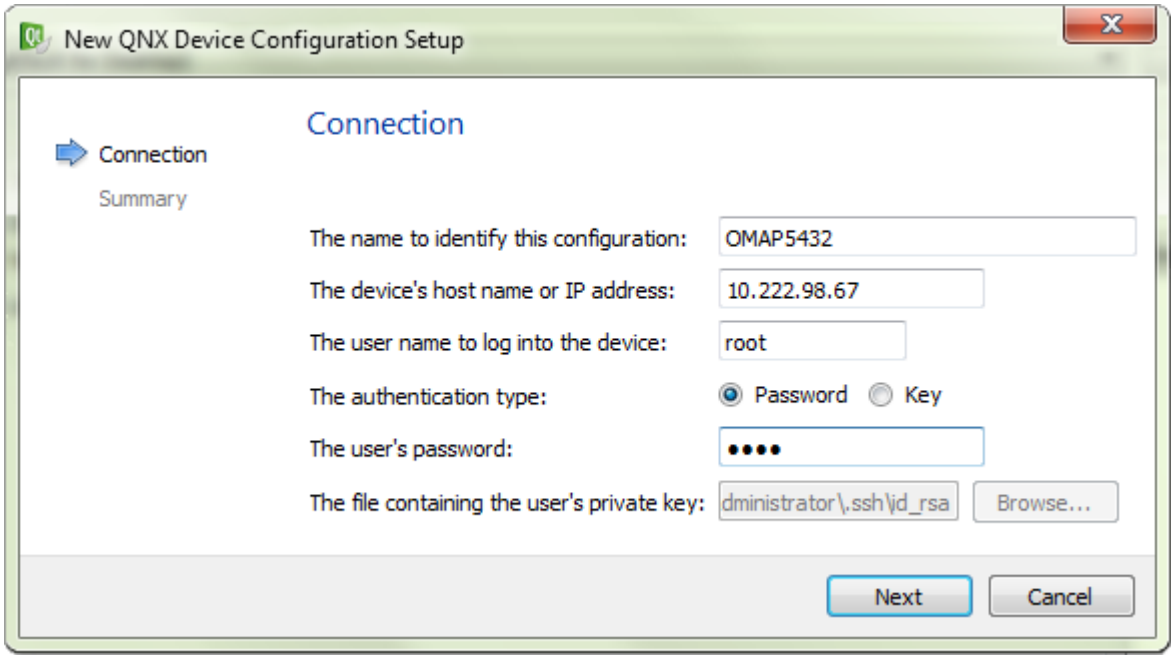
   Initially, Qt Creator shows the default device of `Local PC` in this dialog, because you haven't added a device that represents a QNX target:



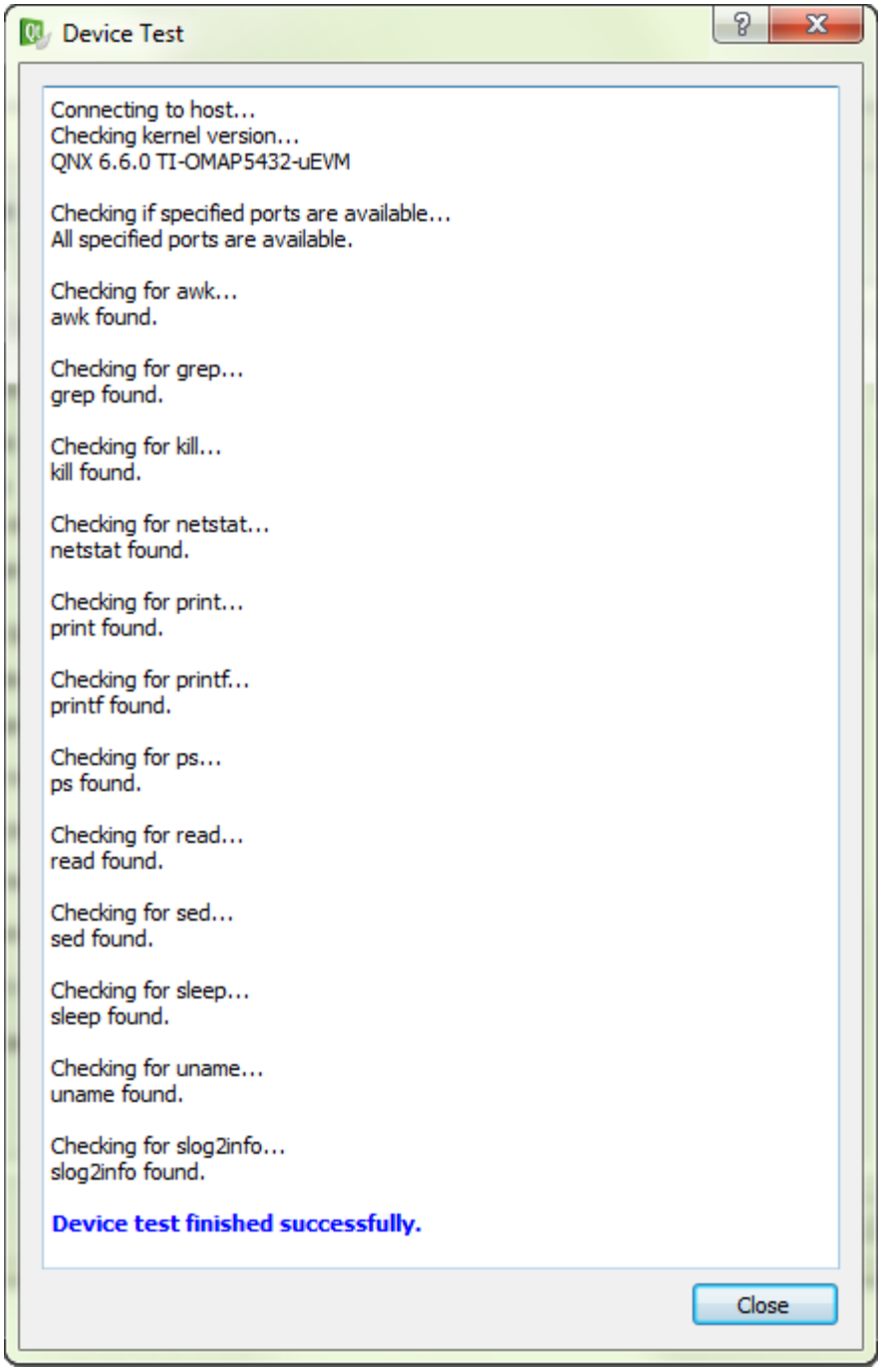
3. In the **Device Configuration Wizard Selection** dialog, choose `QNX Device` and click **Start Wizard**.
4. In the **New QNX Device Configuration Setup** dialog, fill in the connection fields:
   a) Name the device configuration something meaningful, like `OMAP5432`.
   b) Enter the IP address of 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.

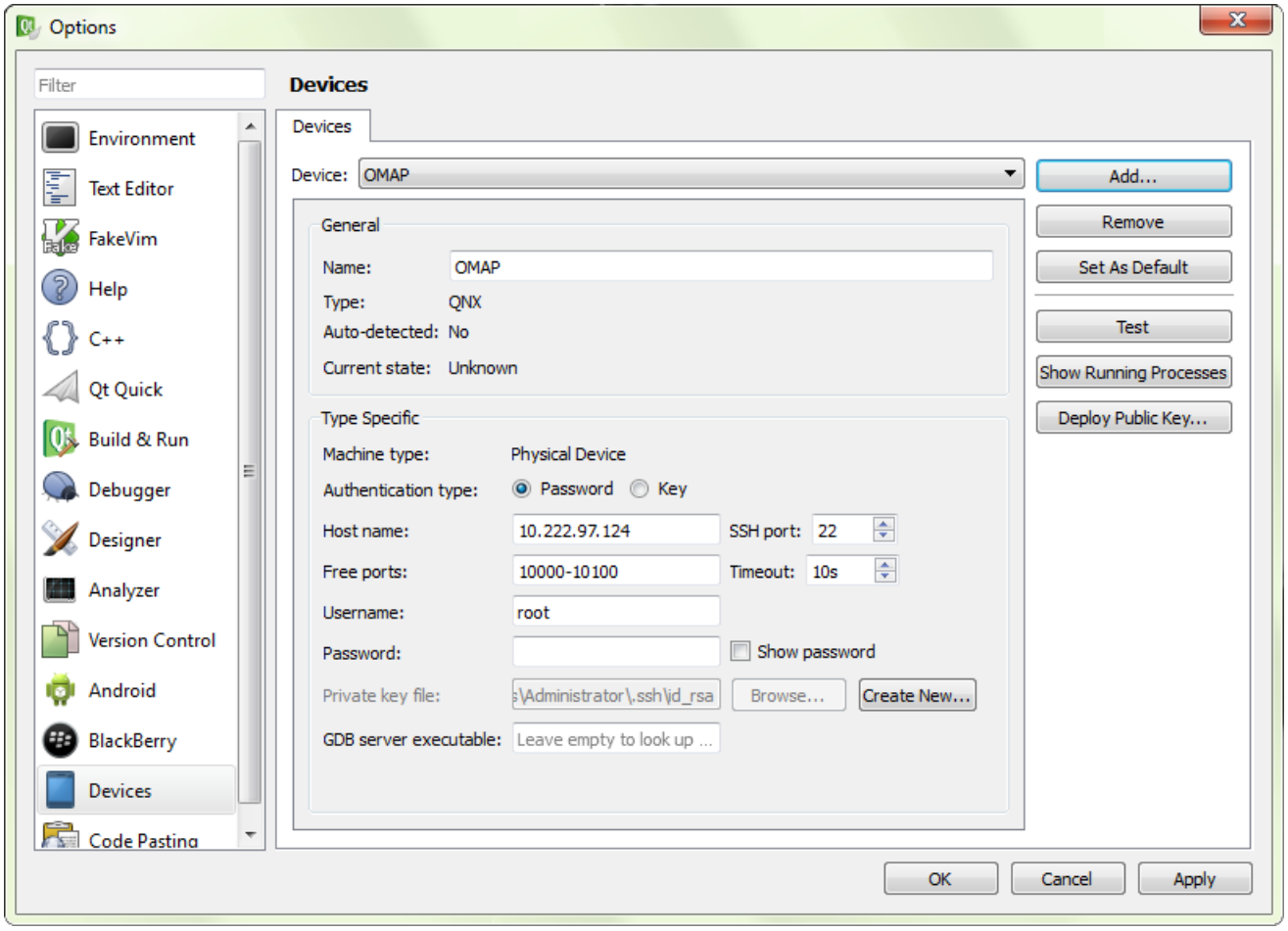
   d) Click **Next**.



5. On the summary page, click **Finish**.
   Qt Creator creates the new device configuration and runs the device connectivity test, which entails connecting to the device and checking if the specified ports and certain key services (e.g., **grep**, **awk**) are available. The test results are shown in the **Device Test** dialog:



6. After examining the test results, click **Close** to return to the **Options** dialog (which now displays the settings of the QNX device).



7. If the connectivity test failed, review the new device's connection settings (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, as shown in Step *5* (p. 17)).

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

CAUTION: Clicking **Apply** isn't enough to save the new device configuration. 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. This is a known issue in Qt Creator.

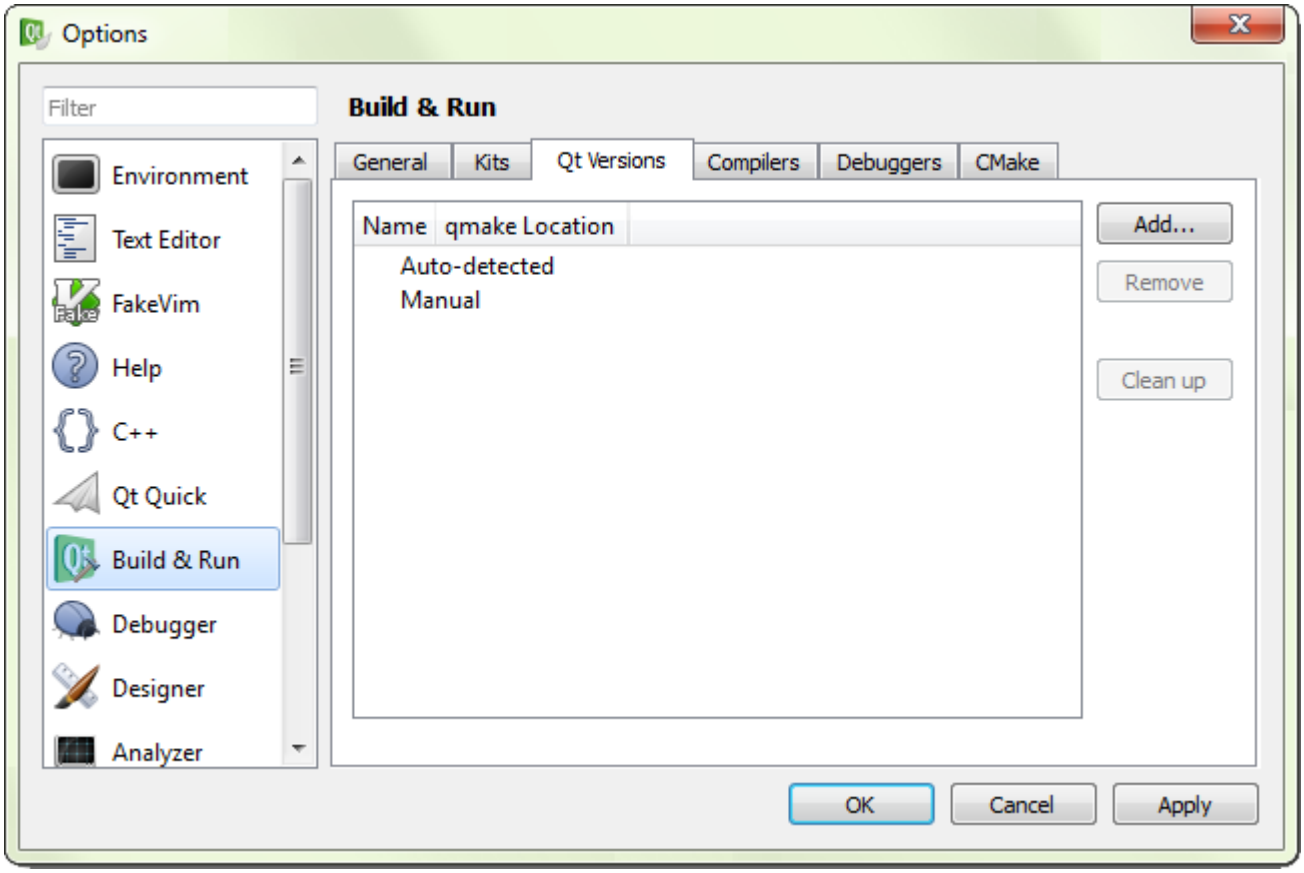
Qt Creator has added a device profile representing your target system. You can now configure a toolchain.

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:

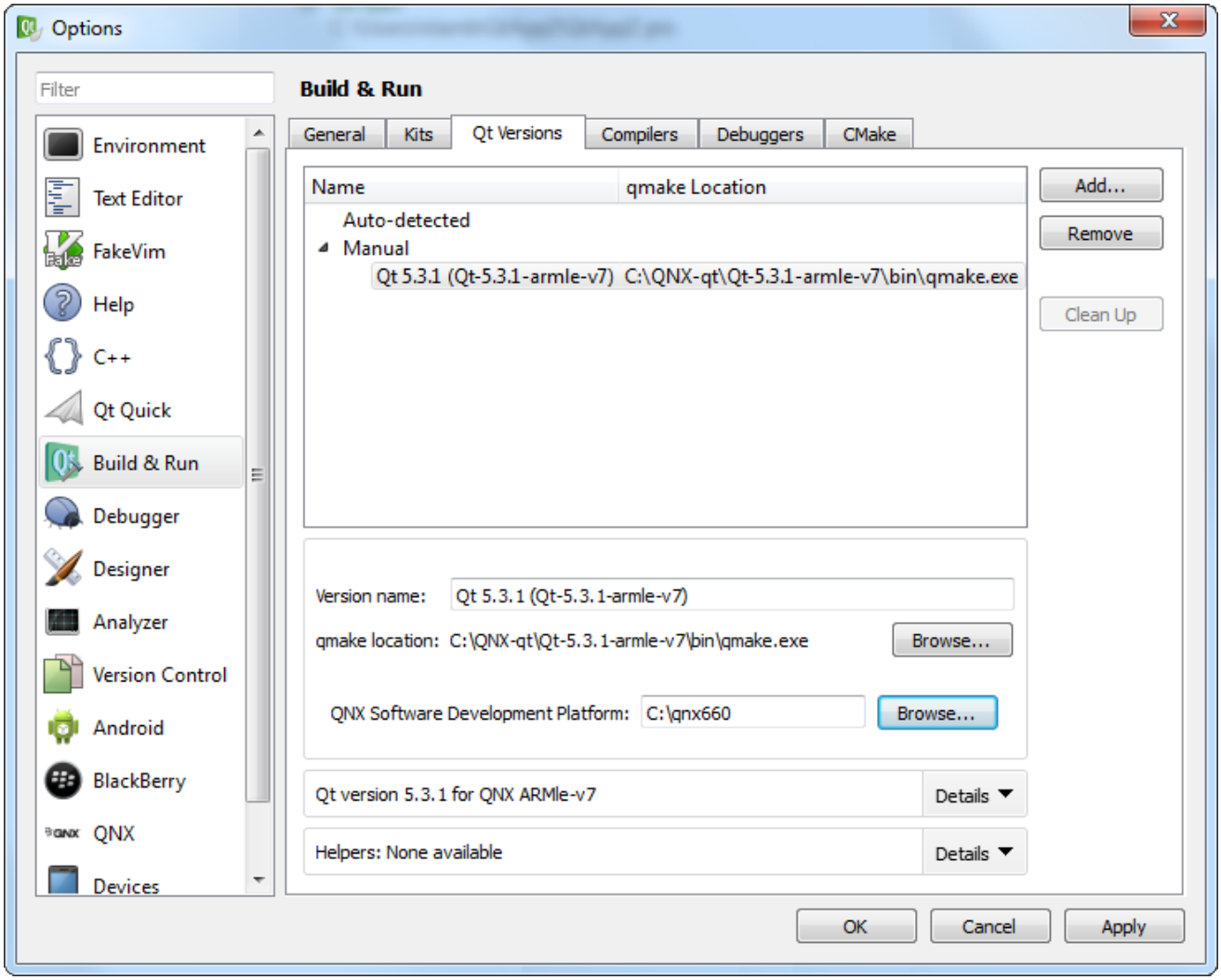
1. In the IDE, select the **Tools** menu, then click **Options** to open the **Options** dialog.
2. Choose **Build & Run** in the left-side menu, click the **Qt Versions** tab in the main viewing area, then click the **Add...** button on the right side.



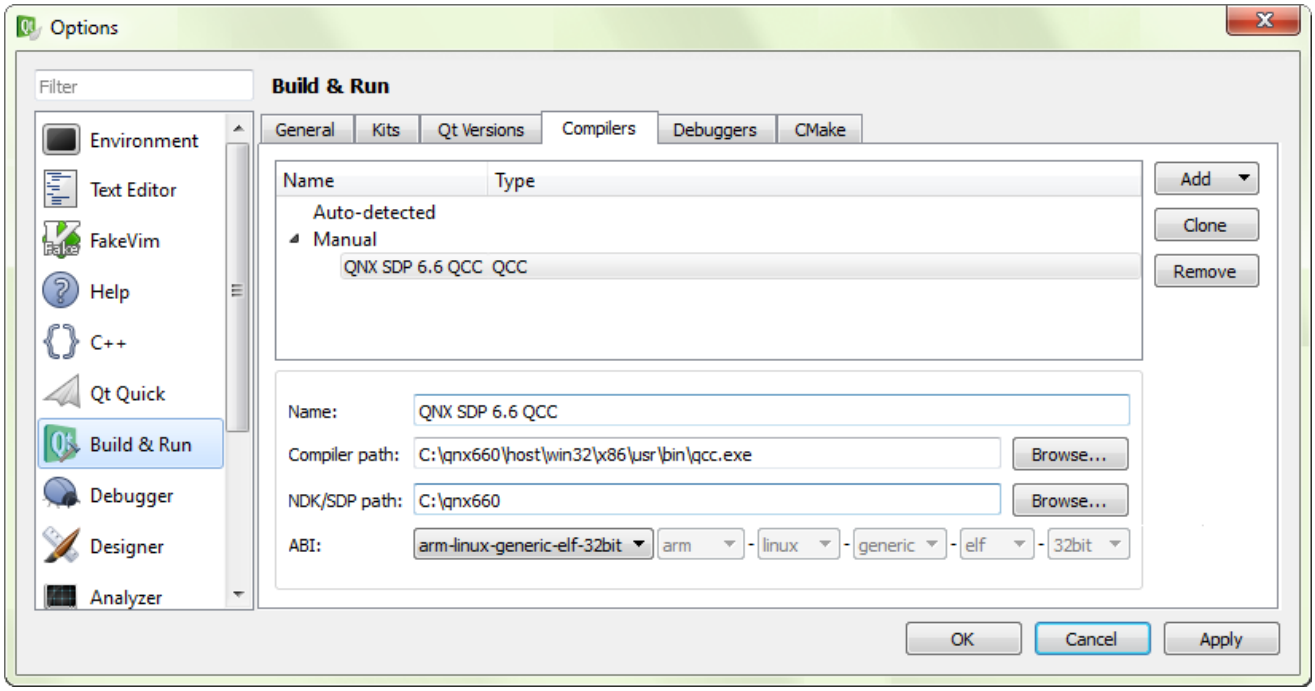
3. In the file selector shown, navigate to the host directory containing the **qmake** version that you're using, select **qmake.exe** (on Windows) or **qmake** (on Linux), then click **Open**.

   The directory containing this utility is ***QT_BASEDIR*\Qt-5.3.1-*variant*\bin**, where *variant* is **x86** or **armle-v7**; on Linux, the directory separators would be forward slashes (`/`).

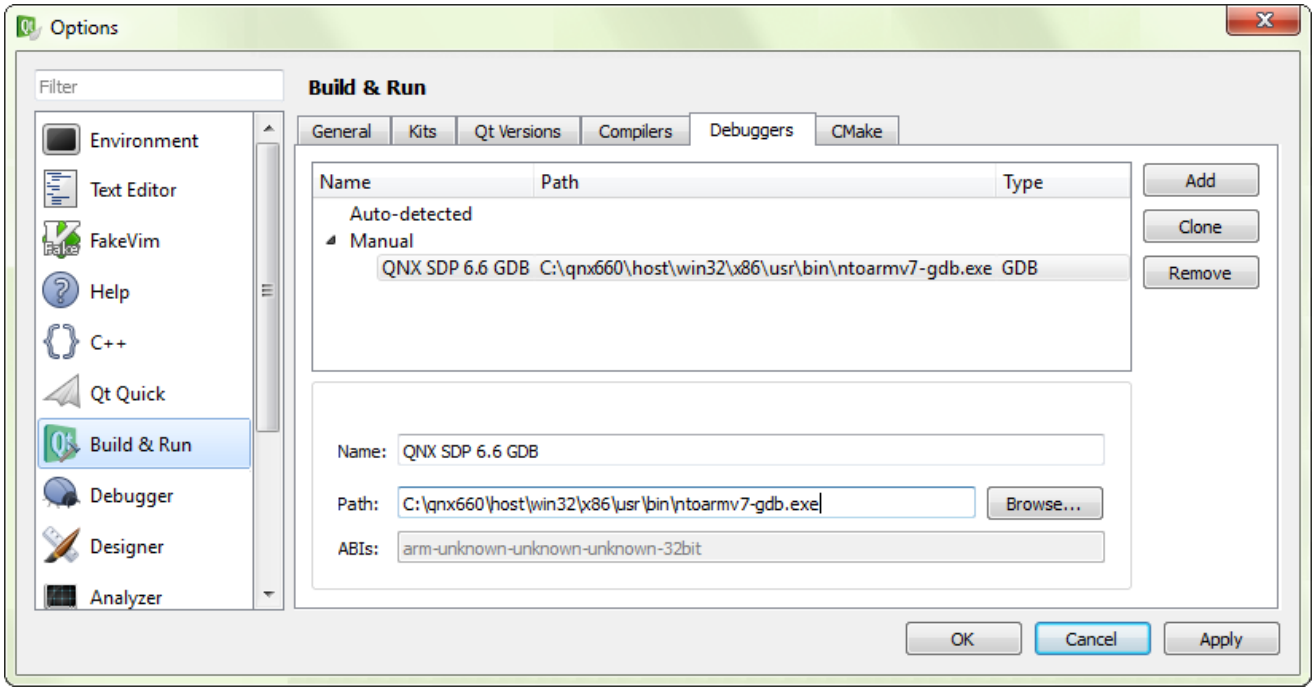
   The **Options** dialog then displays additional fields for configuring the selected Qt version.
4. At the bottom of the dialog, on the line that reads `QNX Software Development Platform`, click **Browse...**.



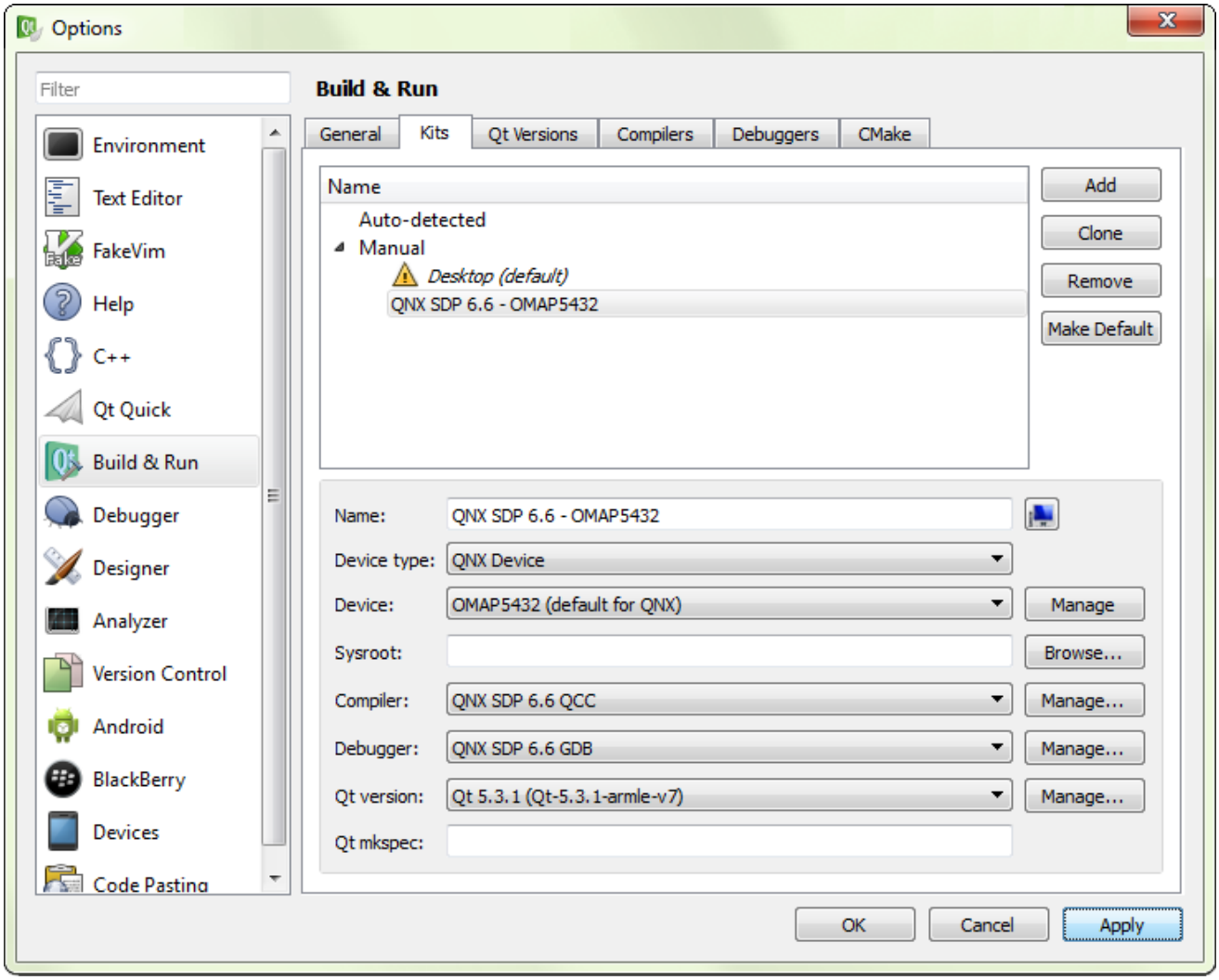
5. In the file selector that the IDE displays, navigate to the SDP installation location (referred to as *DEFAULT_SDP_PATH* in this document) and click **Select Folder**.
   The `QNX Software Development` field now lists the directory containing the QNX SDP 6.6 installation on your host system.
6. Click 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 **Open** 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`.



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 **Open** to confirm the setting.



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 5.3.1 (Qt-5.3.1-armle-v-7)`.



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 Apps and Media systems! 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

Qt Creator supports the entire Qt app lifecycle, from project creation to source file and resource definition to app deployment on a target system.

The sections that follow provide a walkthrough of writing the code for a Qt app, packaging the app, deploying it on a target system, and running it. Here, *app* refers to a Qt program packaged as a Blackberry ARchive (BAR) file, which you can unpackage on the target to make the app accessible from the HMI. To run the app, you simply tap its icon in the Home screen.

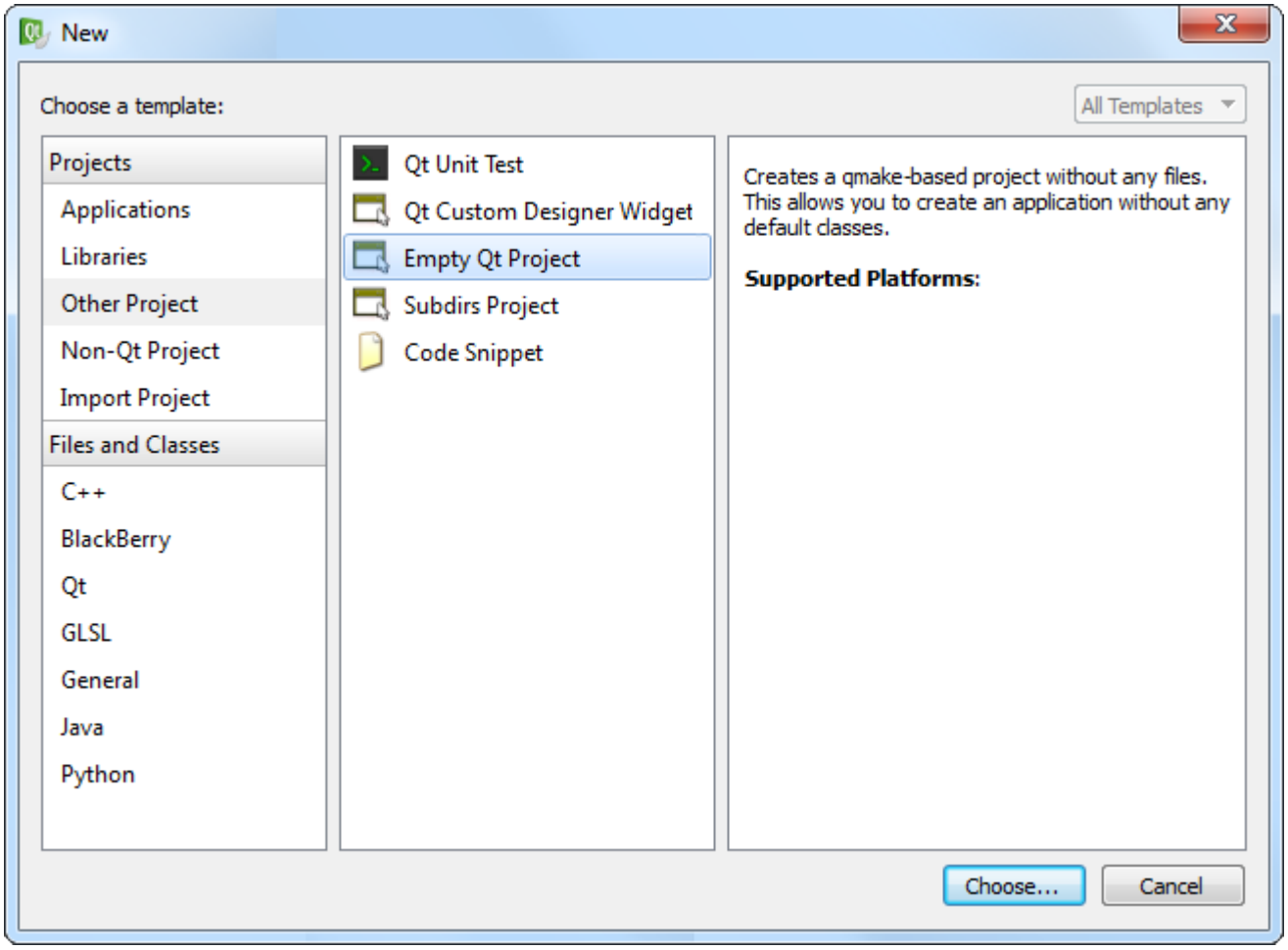
To develop Qt apps, you must have installed and configured the necessary Qt tools (including Qt Creator), as explained in *Preparing your host system for Qt development* (p. 13).

Creating a project for a Qt App

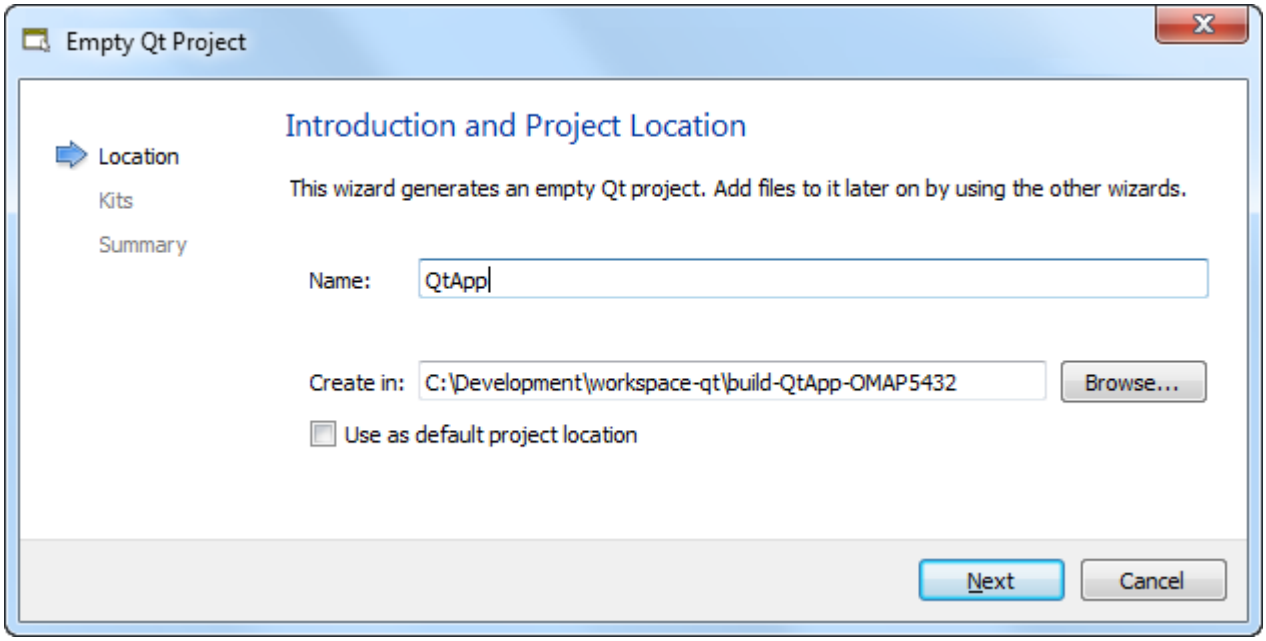
The first task in writing a Qt App is to create a project in Qt Creator and add the necessary components such as the UI definition file, main source file, and an icon.

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 Project**, then **Empty Qt Project**, and then click **Choose...**



4. In the **Location** page of the **Empty Qt Project** dialog, name the project `QtApp`, then click **Next**.



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**.

   To define a kit, you must first define toolchain settings (e.g., compiler, debugger), as explained in "*Configuring a toolchain in Qt Creator* (p. 20)".

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

Defining the UI

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

To define the UI:

1. Click the **Edit** icon on the left side, right-click the `QtApp` folder in the **Projects** view, then choose **Add New...** in the popup menu.
2. In the **New File** dialog, select `Qt` in the **Files and Classes** list, then `QML File (Qt Quick 2)` in the list of file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the **New QML File** dialog, 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 box displaying `Hello World`.

The QNX Apps and Media reference image has a similar HTML5 sample that displays “Hello World” but here, we're writing an app with a basic UI to demonstrate Qt app development and deployment. In fact, you can replace the QML code here with whatever you like to display a different UI.

6. Save the file.

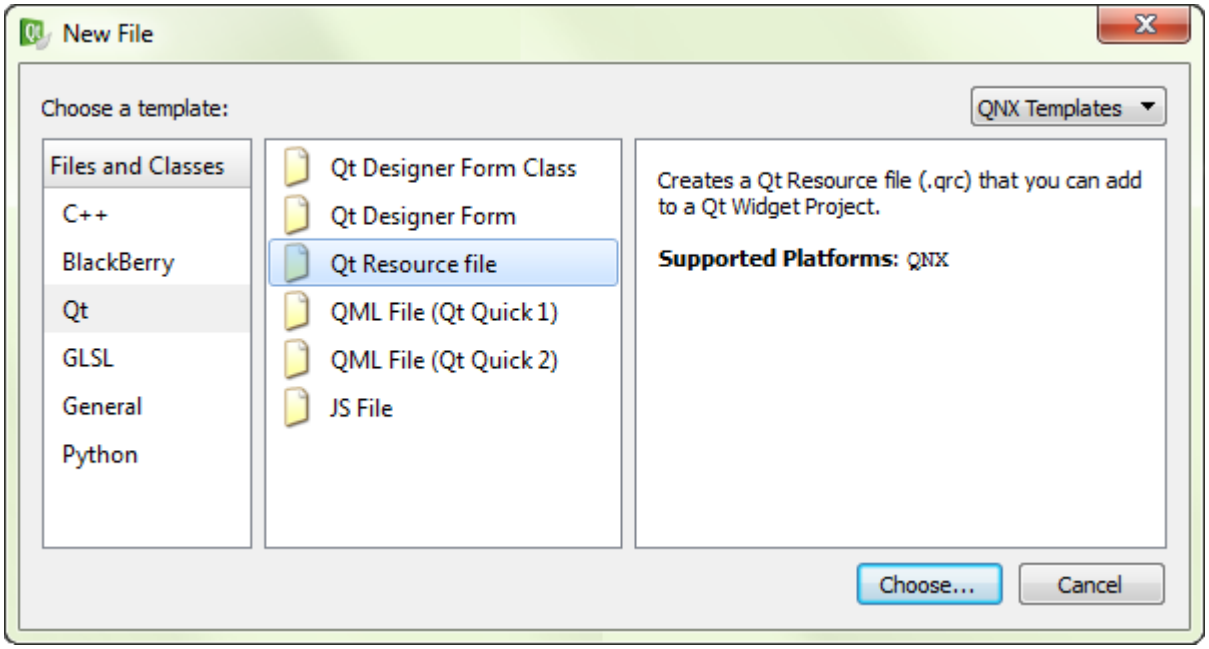
Making a QML file into a project resource

After you've defined the UI in a QML file, you can create a Qt resource file that includes the QML file and then add this resource file to your project. This makes Qt Creator include the UI definition in the binary file.

There are several ways to access resources in Qt apps running on a QNX Apps and Media system. In addition to compiling resources into their binaries, apps can access resources from within their 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.

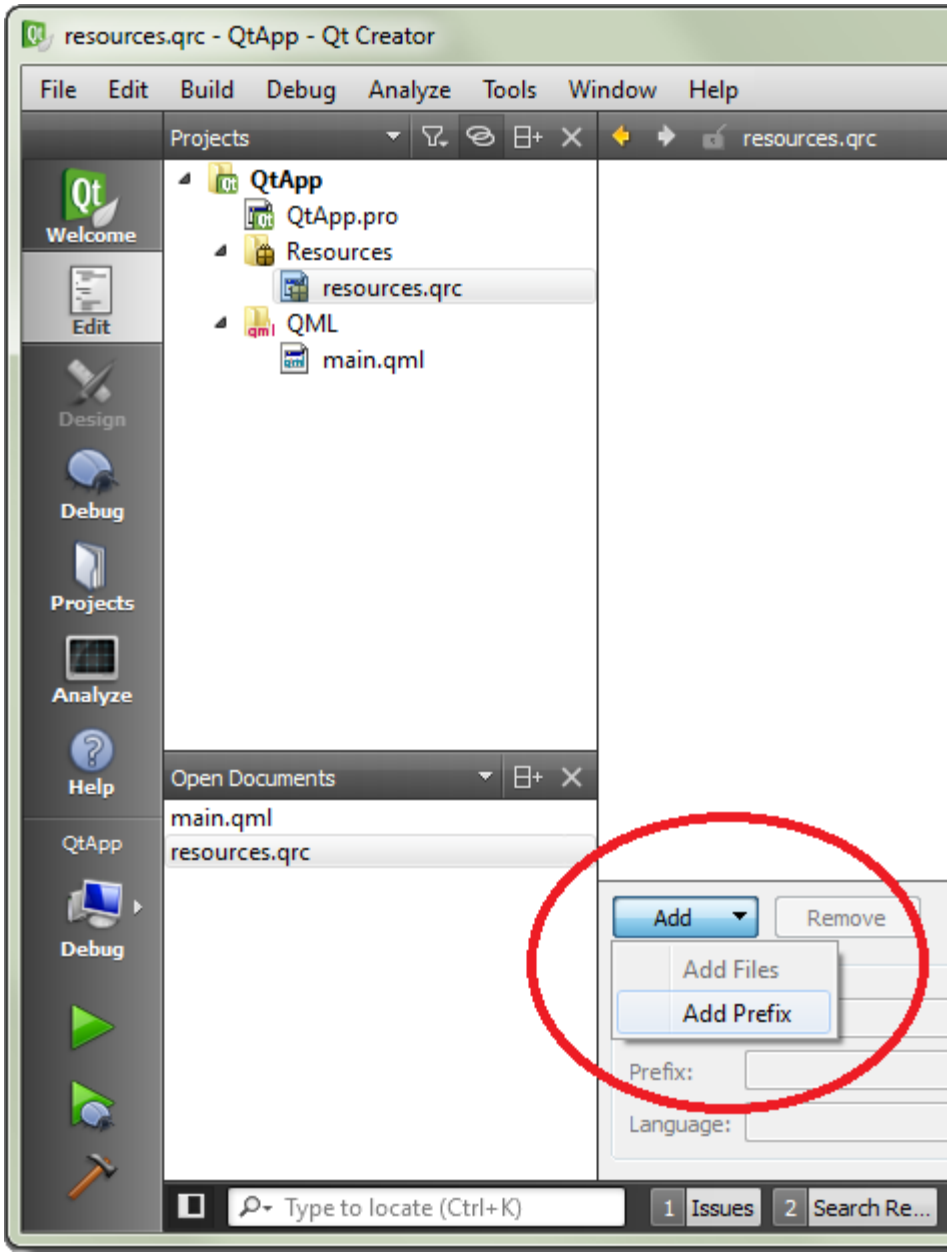
To make the UI-defining QML file into a project resource:

1. Click the **Edit** icon on the 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 file types (shown in the middle), then click **Choose...**



3. In the **Location** page of the **New Qt Resource file** dialog, 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 configuration area near the bottom, click **Add**, then choose **Add Prefix**.



6. In the **Prefix** field, replace the default text with `ui`.
7. Click **Add** again, then choose **Add Files**.
8. In the file selector that Qt Creator opens, navigate to the project directory and select `main.qml`, then 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++ 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 file types (shown in the middle), then click **Choose...**
3. In the **Location** page in the resulting dialog, 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 project file (**QtApp.pro**) for editing and add this 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.

The project file can define many variables that affect how **qmake** builds the project; for the full list, see the *Variables | QMake* reference in Digia's online Qt documentation.

Adding an image for the app icon

To provide an icon that lets users identify and launch your app in the target HMI, you can save an image file in your project folder.

To add an image to use as the app icon:

- Copy the following image and save it as **icon.png** in the `QtApp` project folder:

We provide a sample icon here for convenience, but you can use any appropriately sized image as an icon.

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

Writing the app descriptor file

After your project is set up, you can package the Qt app in a BAR file so it can be deployed onto a QNX Apps and Media target. 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 the 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 file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the **New Text file** dialog, 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">
    <id>com.mycompany.QtApp</id>
    <name>Qt App</name>
    <versionNumber>1.0.0</versionNumber>
    <description>The Hello World Qt demo app.</description>
    <category>demo</category>
    <icon>
        <image>icon.png</image>
    </icon>
    <buildId>1</buildId>
    <author>My Company Inc.</author>
    <permission system="true">run_native</permission>
    <env var="QQNX_PHYSICAL_SCREEN_SIZE" value="150,90"/>
    <asset type="Qnx/Elf" path="QtApp" entry="true">QtApp</asset>
</qnx>
```

   The app-descriptor file defines the app name, description, icon file, and other fields that contain authoring information and settings for the initial window. It also sets the *QQNX_PHYSICAL_SCREEN_SIZE* environment variable, which defines the height and width of the app's display area. Finally, the app descriptor file also provides asset information, including the binary file path and format.

Environment variables

In the app descriptor file, you can define environment variables that your app can access from its sandbox environment. In our sample Qt apps, these variable settings define the logging level or the app's physical display area, but you can set any variable you want.

Environment variables are set using `<env>` tags, where the `var` attribute lists the variable's name and the `value` attribute lists its value:

```
<env var="QQNX_PHYSICAL_SCREEN_SIZE" value="150,90"/>
```

Physical display area

The *QQNX_PHYSICAL_SCREEN_SIZE* variable defines the height and width of the app's display area on the screen. The width is listed first, followed by a comma, followed by the height. Note that the dimensions are specified in millimeters, not pixels. This is because the QNX Apps and Media target requires a physical unit and not a virtual unit.

We strongly recommend setting this variable to better control how your app is shown in the target HMI. If you don't define this variable, the display size defaults to 150 mm by 90 mm, which may not be optimal for viewing your app. Also, you'll receive a **stdout** warning when starting your app, although it will still run.

Library paths

The *LD_LIBRARY_PATH* variable should *not* be used to define the path of dynamic libraries used by your app. When defined in your project, this variable setting overrides the system setting on the target. The target environment must be configured so all essential libraries, including Qt and other commonly used libraries, are visible to the dynamic linker. For instance, on the target, *LD_LIBRARY_PATH* may be set to:

```
lib:/usr/lib:/usr/qt5-5.3/lib
```

Suppose you override this variable in the project for a Qt app so that it can access certain libraries in its sandbox, as follows:

```
<env var="LD_LIBRARY_PATH" value="app/native/lib"/>
```

In this case, your app won't start on the target because the dynamic linker won't be able to find the Qt libraries or any shared libaries outside of the **app/native/lib** path needed by the app. While you could expand the project variable setting to include all the paths in the target's LD_LIBRARY_PATH value, this depends on you knowing the target's setup, which might not be the case if you're developing apps for a third party. Also, if the target setup changes, you would have to update your project settings.

We recommend that you instead define the **RPATH** link option in your project to give the app access to the libraries included in its sandbox environment.

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
`<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>`
`<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. Unless otherwise noted, the attributes are optional.	**`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*. **`dest`** The asset's destination path. Typically, the value is the last part of `path` (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 working directory of the packager.	`<asset type="Qnx/Elf" path="QtApp" entry="true">QtApp </asset>`

Name	Required	Description	Attributes	Example
			public When true, store the asset in the public directory of the BAR file, so it's readable to other applications. Icon assets should be public. The default setting is false. **type** The asset type. For Qt binaries, use `Qnx/Elf`.	
`<author>`	No	Specifies the author name (typically the company or developer name).		`<author>` `My Company Inc.` `</author>`
`<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>`
`<category>`	No	Indicates the category to which the application belongs.		`<category>` `media` `</category>`
`<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>`

Name	Required	Description	Attributes	Example
`<entryPointType>`	Not if the entry point is defined in an `<asset>` tag; otherwise, yes.	Defines the entry point type, which can be either `Qnx/Elf` (for native applications, including Qt applications) or `Qnx/WebKit` (for applications based on HTML5 and Cordova).		`<entryPointType>` `Qnx/WebKit` `</entryPointType>`
`<env>`	No	Defines environment variable settings. For Qt apps, we recommend defining the *QQNX_PHYSICAL_SCREEN_SIZE* variable, but you can define others as well, as explained in "*Environment variables* (p. 34)".	**`var`** *(Required)* Name of the environment variable. **`value`** *(Required)* Value of the environment variable.	`<env var="QQNX_` `PHYSICAL_SCREEN_SIZE"` `value="150,90"/>`
`<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 represented by an empty spot in the viewing area showing the installed apps.		See the *`<image>`* (p. 37) element.
`<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>`
`<name>`	Yes	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>`

Name	Required	Description	Attributes	Example
`<permission>`	No	Specifies the privileges (also known as capabilities, user actions, or actions) that the application requests from the OS. The permission settings relevant to Qt apps are listed in "*App permissions* (p. 40)".	**`system`** *(Required)* Specifies whether the action is a system (not a user) action. For Qt apps, this attribute must be set to true.	`<permission system="true">` `access_internet` `</permission>`
`<platform Architecture>`	No	Specifies the processor type 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`—compile your application to run on a simulator • `armle-v7`—build the application to run on a device		`<platformArchitecture>` `x86` `</platformArchitecture>`
`<platform Version>`	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>`
`<qnx>`	Yes	Defines the top-level element of the schema used for the app descriptor file.	**`xmlns`** *(Optional)* URL referencing the XML namespace.	See the example of the app descriptor file in "*Writing the app descriptor file* (p. 33)".
`<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. These hyphenated strings are based on the IETF Best Current Practice (BCP) 47 specification (e.g., `en-US` for U.S. English, `de-DE` for German, or `fr-CA` for 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>`

Name	Required	Description	Attributes	Example
`<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 an upgrade is required. 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

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

The permissions listed here are those that apply to Qt apps. Other app types (e.g., Android) may have different permissions.

For Qt app decriptor files, each `<permission>` element can define of the following one permissions:

Permission element value	Description
`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.
`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, and Wi-Fi data.
`access_shared`	Allows the app to read and write files shared between all apps. When this permission is 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.
`configure_system`	Enables the app to modify system settings, including Bluetooth, Wi-Fi, network connection, and software update settings.
`manage_cert`	Grants the app access to browser certificates. This is needed to browse content using HTTPS and to save certificates locally.
`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.
`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.
`record_audio`	Grants the app access to the audio stream from a microphone attached to the system.
`set_audio_volume`	Allows the app to control the audio volume.
`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.
`use_installer`	Enables the app to access the `appinst-mgr` native service, which provides access to the install and uninstall mechanism.

Building the app

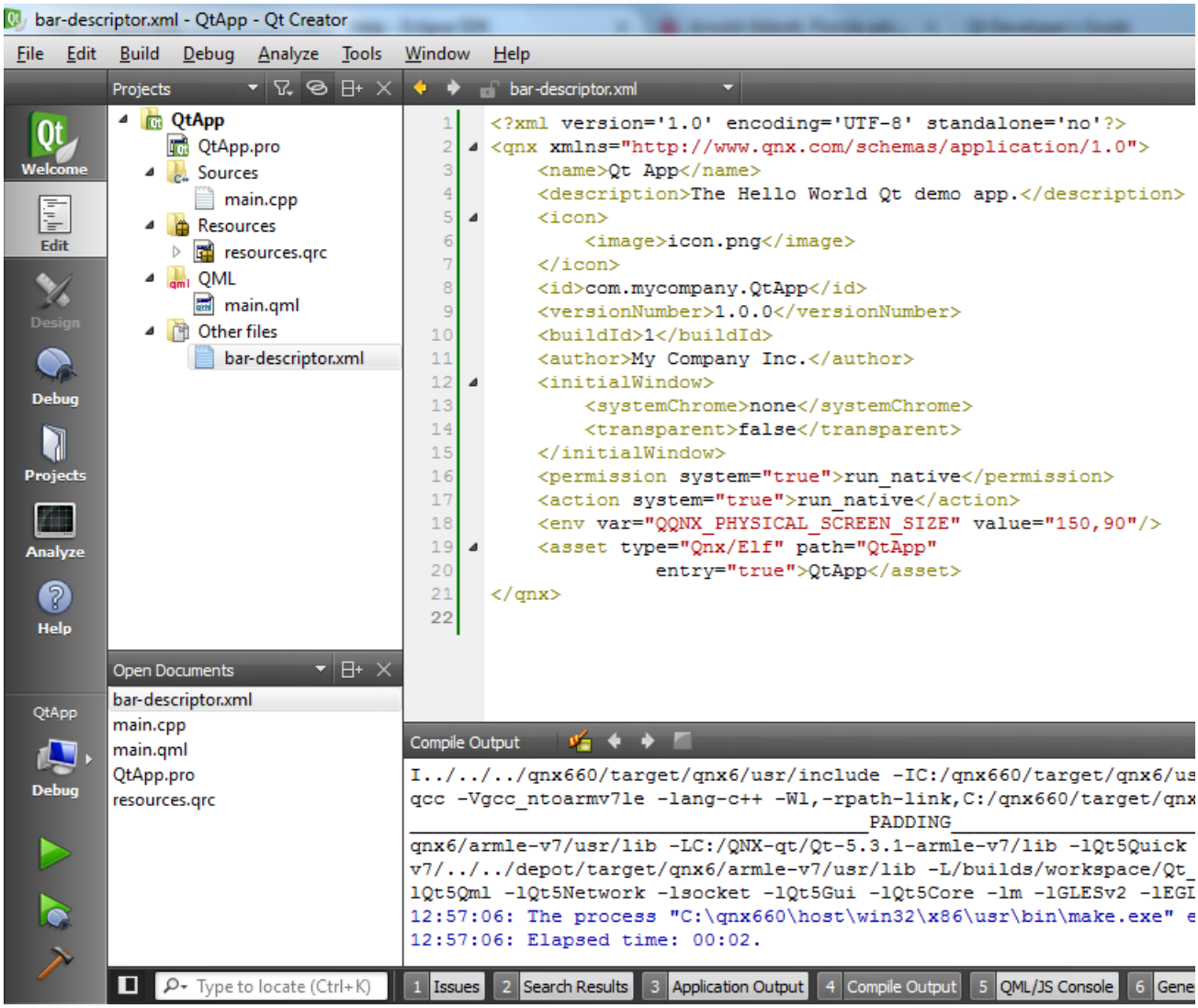
After creating the Qt project and defining the resources for the app, you can build its binary to verify the correctness of the code and the project configuration.

Qt Creator has many features to make compilation and debugging easier, as explained in "*Tips for compiling programs in Qt Creator* (p. 42)".

To compile the app:

- In the **Build** menu, choose **Build Project "QtApp"**.

  Qt Creator starts building the application and displays the QCC output in the **Compile Output** window.



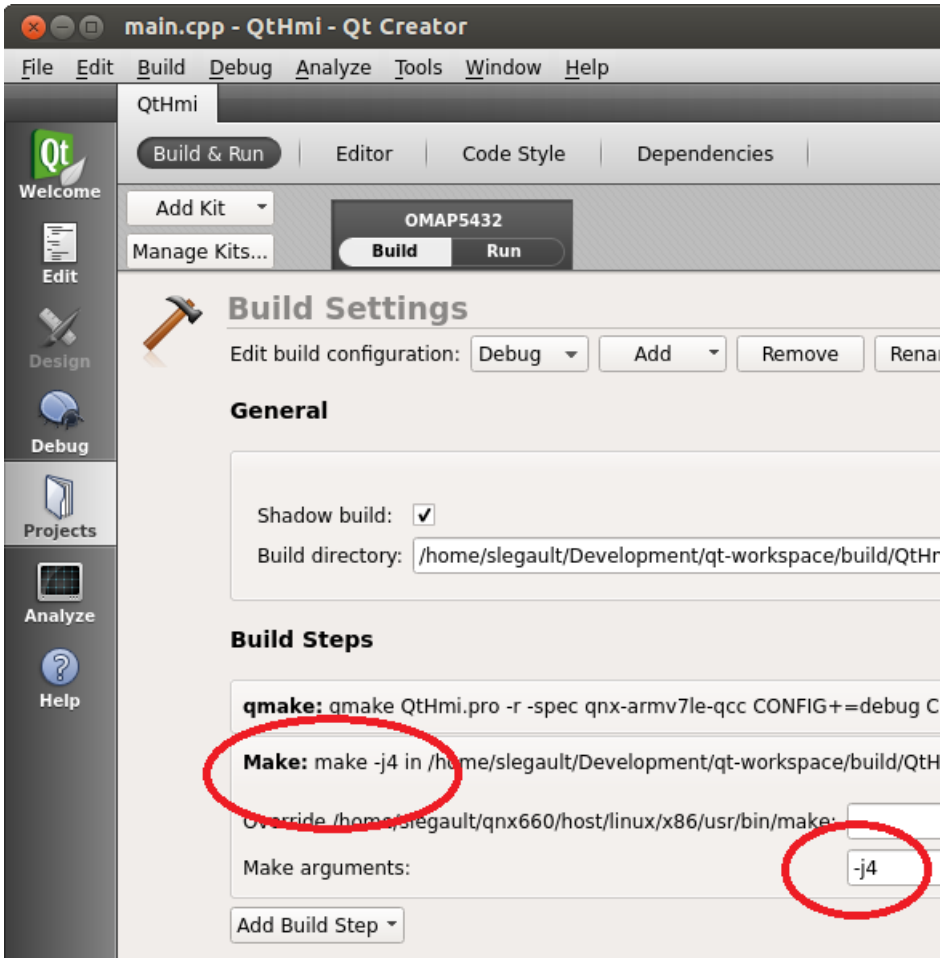
If the application builds successfully, the binary will be in the build directory specified in the **General** section of the **Build Settings** page, which is accessed by clicking the **Projects** icon on the left side and then selecting the QtApp project.

If the build fails, you can review the messages shown in the **Compile Output** window (which is accessed by clicking the button with the same name at the bottom) to determine the cause of the failure and then take corrective action to fix the project.

Tips for compiling programs in Qt Creator

The following actions can help you compile and debug programs efficiently:

- To see the compilation output when your project gets built, click the **Compile Output** button on the bottom of the screen. This displays the output of the QCC compiler. While the **Issues** view provides a summary of any problems encountered during compilation, the **Compile Output** view shows more information that helps explain the cause of an error listed in **Issues**.
- To speed up compilation, you can inform Qt Creator of the number of CPU cores on your host machine. To do this, select the **Projects** tab, go to the **Build Settings** page, and locate the **Build Steps** section. You can then expand the **Make** instruction and in the arguments field, add `-j` *n*, where *n* is the number of cores on the machine:



This action instructs `make` to use multiple threads during compilation, which can significantly reduce build times for large projects. (It won't make any difference for our small sample project but does help when building large applications.)

In some Windows versions of `make`, the `-j` option isn't implemented and so it has no effect.

- If you encounter compilation problems related to `moc` or `vtables`, clean your project (by selecting **Build → Clean All**), rerun `qmake` (by selecting **Build → Run qmake**), and then rebuild your project.

  Qt Creator uses `qmake` to generate makefiles containing instructions on how to compile the project. Sometimes the makefiles become out-of-date and must be manually regenerated by doing those previous actions.

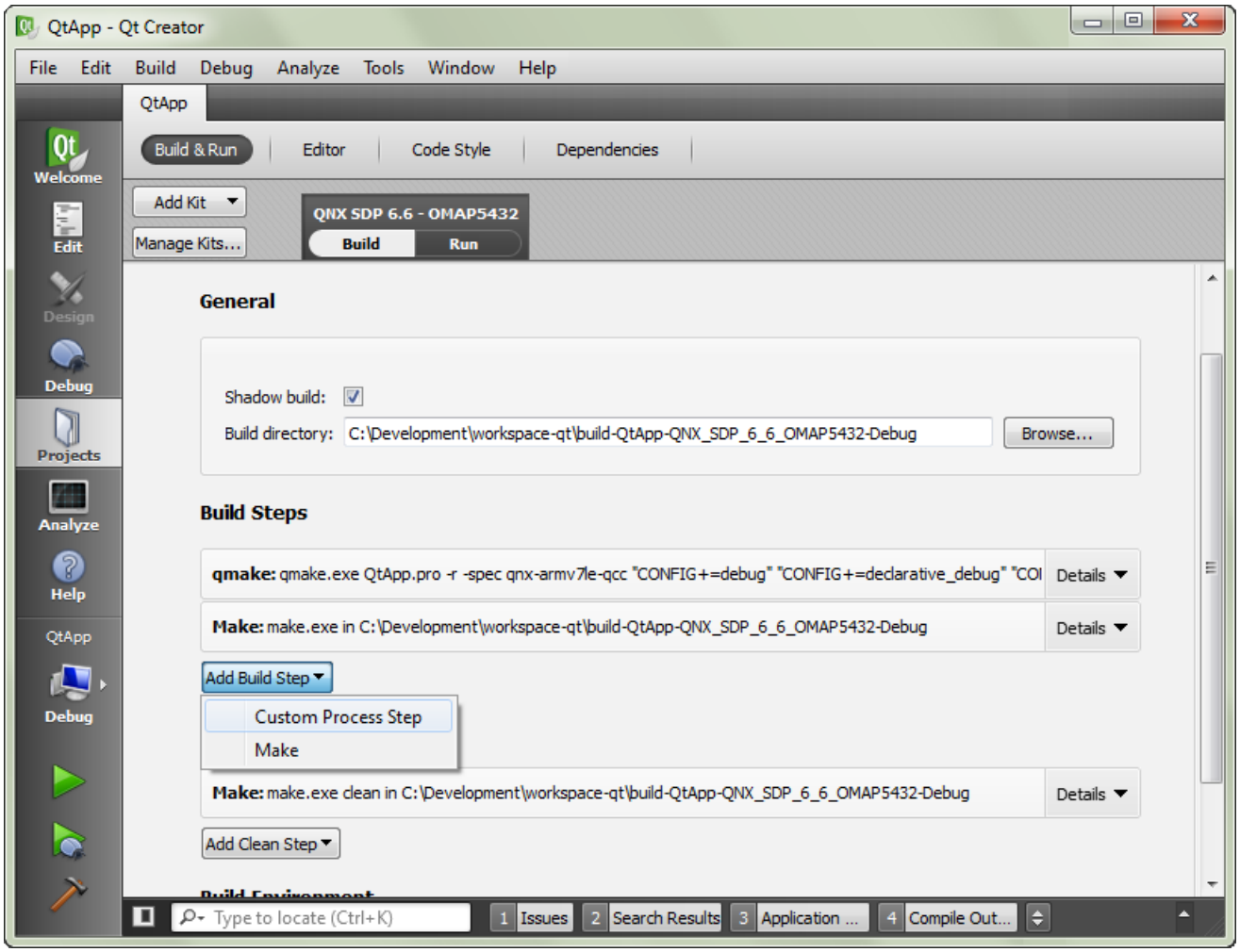
Packaging the app into a BAR file from Qt Creator

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. 47). 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 the 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:

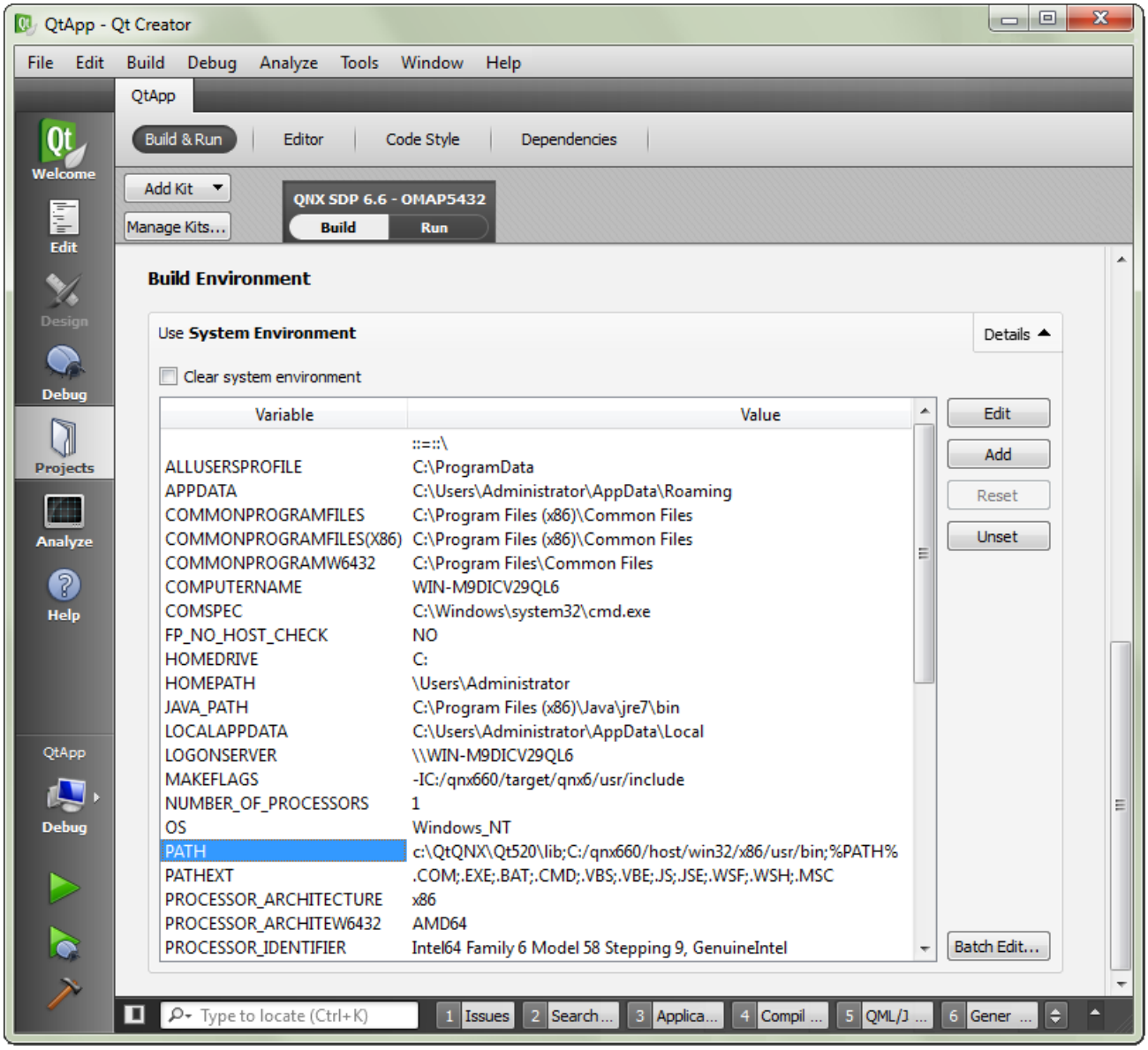
```
QtApp.bar %{sourceDir}\bar-descriptor.xml QtApp -C %{sourceDir}
 %{sourceDir}\icon.png
```

While the above command may appear across multiple lines in your viewer, you must enter it on one line in Qt Creator. Also, the directory separators in this example are backslashes (`\`), which are used for Windows, but you must use the appropriate separator for your OS (i.e., `"/"` if you're running Linux).

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. 48)".

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.



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 `blackberry-nativepackager` 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 **Compile Output** window.

The `QtApp` app is packaged in a BAR file and can then 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 Qt environment variables* (p. 34) 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:

- In a BlackBerry 10 OS 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. The order for other command-line arguments 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. 48)".

After your app is packaged, you can deploy it on the target, as explained in "*Deploying the BAR file on the target* (p. 51)".

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. 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 BlackBerry 10 OS terminal connected to the target and the result would be the same.

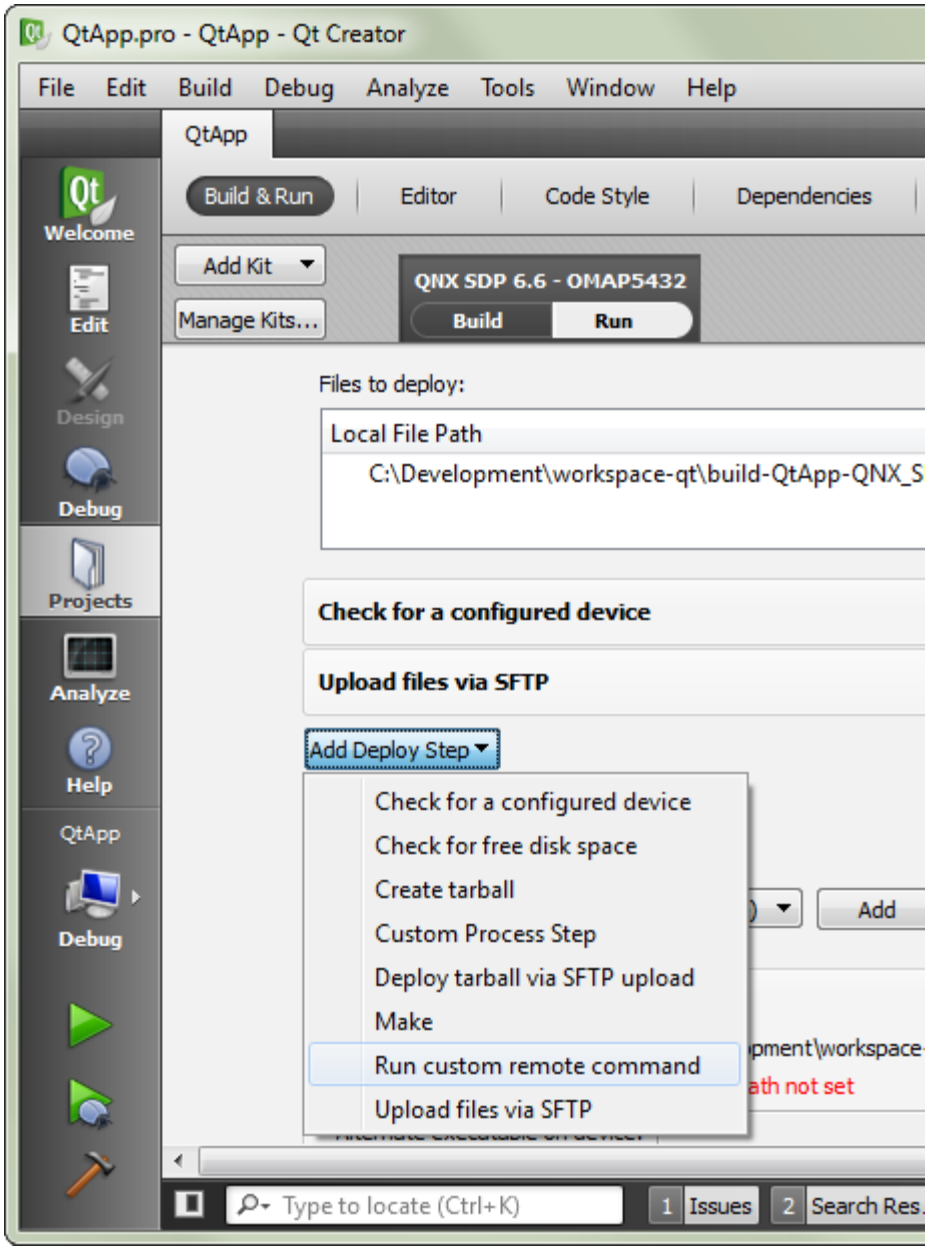
To deploy an app on the target from Qt Creator:

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

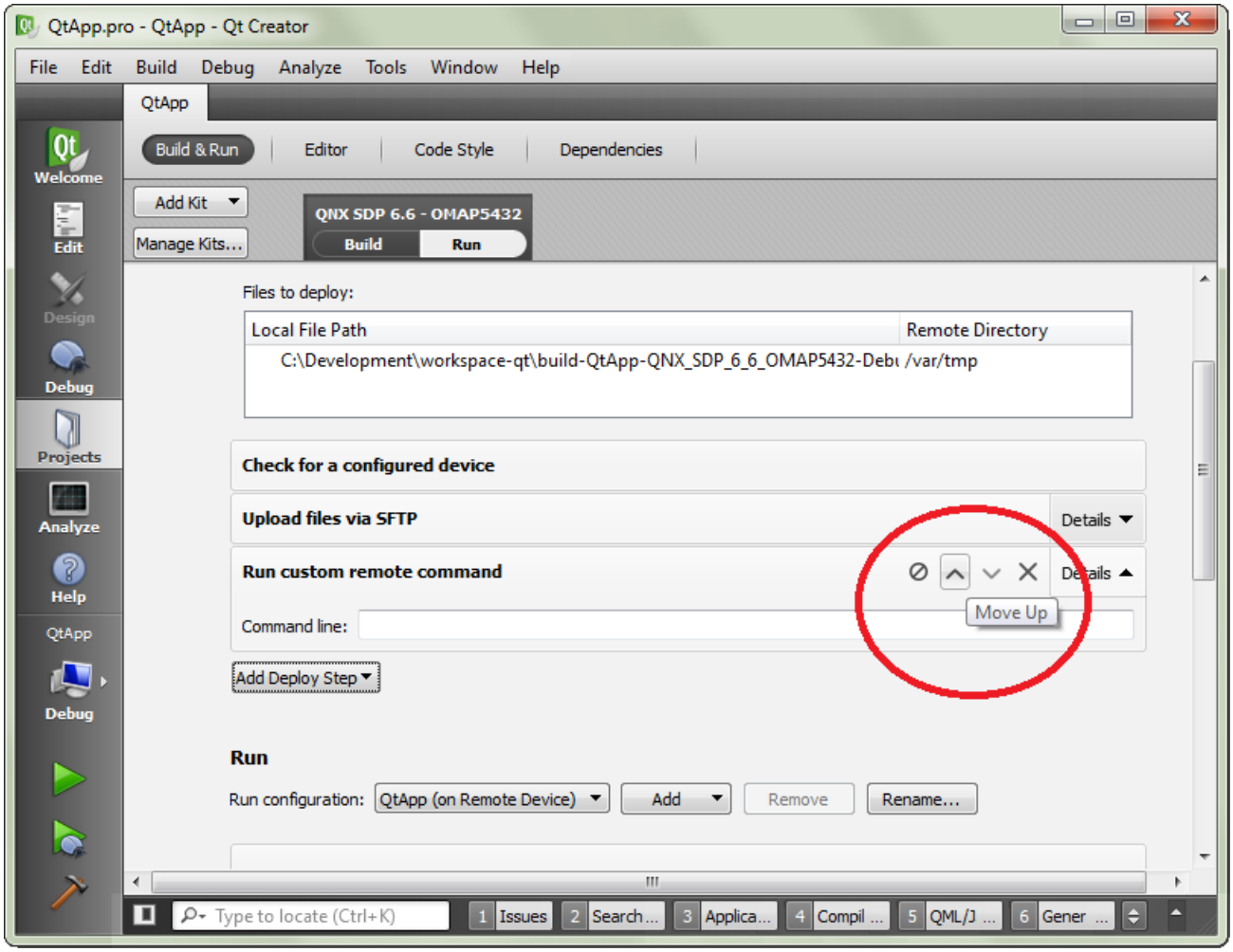
   ```
   barfile.path = /var/tmp
   barfile.files = $$OUT_PWD/QtApp.bar
   INSTALLS += barfile
   ```

   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. 16)".

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



4. In the newly displayed box 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.



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

   ```
   mount -uw /base
   ```

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

6. Click **Add Deploy Step** again, choose `Run custom remote command`, and enter the following command in the newly displayed **Command Line** field:

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

   This command runs the installer on the target, installing the BAR package in a location accessible to the Home screen.

   You should have the following deployment steps (where the first and third were predefined):

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

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

7. 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 to the Home screen app. The IDE displays timestamped messages detailing the outcomes of the deployment steps in the **Compile Output** window.

Running the app

After you've unpackaged the app's BAR file on the target, you can run the app from the target HMI.

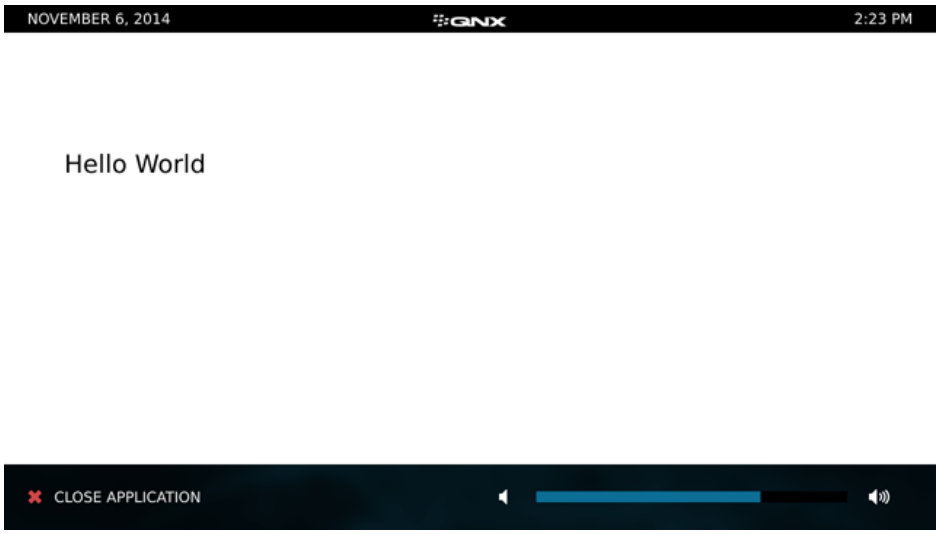
To run the app on the target:

1. Access the Home screen in the HMI.

   You should see a new icon, labelled `Qt App` displayed with the other icons:

2. Tap the `Qt App` 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 specified in the `<initialWindow>` tag, within the physical area defined by the *QQNX_PHYSICAL_SCREEN_SIZE* environment variable (also set in the app descriptor file).

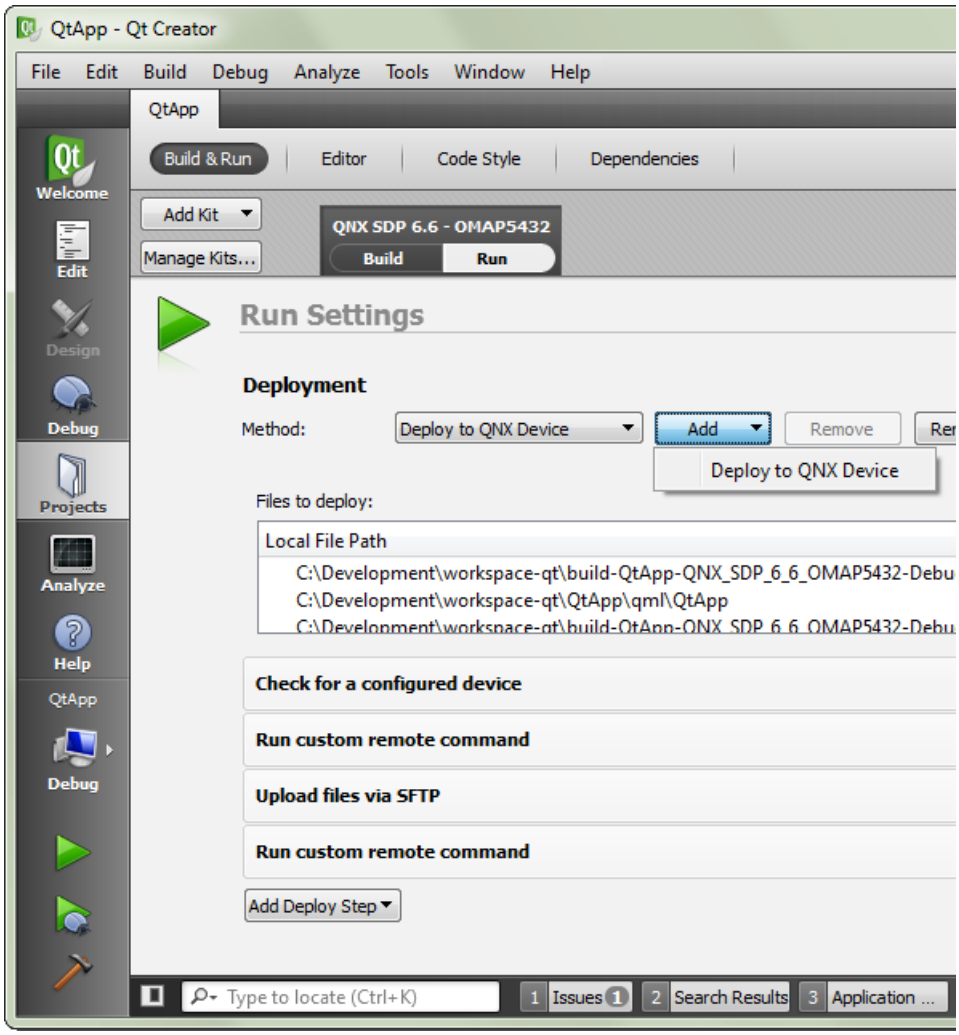
Cleaning the target before redeploying a BAR file

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.

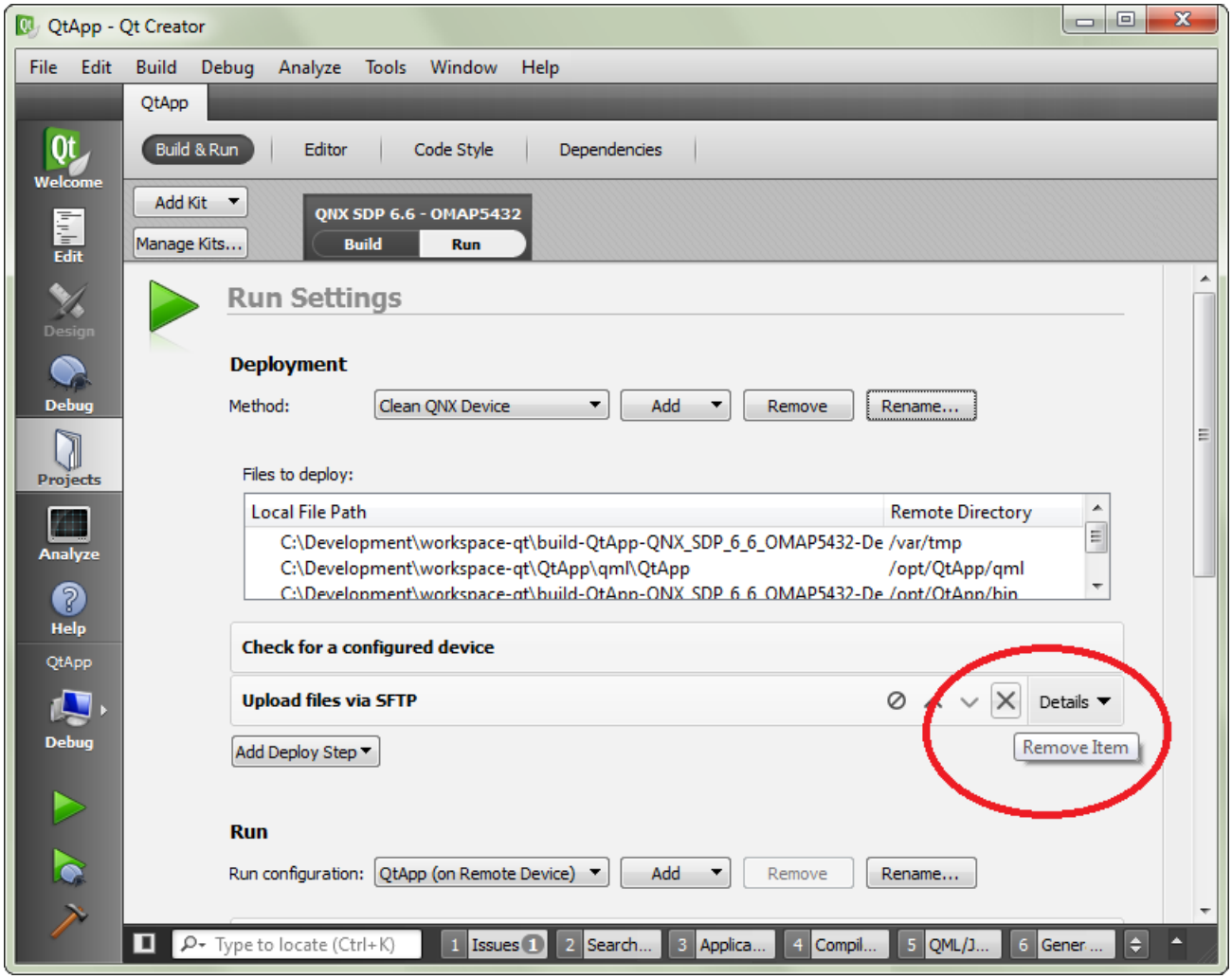
You can also issue these commands manually through a BlackBerry 10 OS terminal connected to the target and the result will be the same.

To clean an app's installation on the target:

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



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`.



4. Click the **Add Deploy Step** button, then choose `Run custom remote command`.
5. In the new **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
Building libraries for Qt apps

When writing applications, it's often necessary to use libraries to store specific functionality (e.g., graphics functions, filesystem access). On QNX targets, apps run in sandbox environments with limited access to system facilities, meaning that their required functionality must be contained in libraries accessible in the sandbox.

In the QNX Qt environment, an app can either statically or dynamically link in its required libraries. With static linking, the app links the static object-code library (**.a**) files into its executable. This strategy ensures that the required library functionality is always accessible to the app. With dynamic linking, the libraries are stored in "shared library" (**.so**) files that are included in the app package. At runtime, the app binary must load these files.

Each app must package all the **.so** files it needs because the separate sandboxes for separate apps mean that apps can't actually share dynamic libraries. Therefore, when a given library is included in an app, the package size increases by the same amount whether the library is statically or dynamically linked. Also, when one of its libaries is upgraded, the app must be repackaged and redeployed.

While static linking is the recommended option, it may not always be possible due to licensing restrictions or other issues. When dynamic linking is the only option, special considerations apply for the sandbox environment. The tutorial that follows demonstrates how to dynamically link a library into an app and then deploy the library as part of the BAR package.

In our example, we will create a "third-party" library for use by our QtApp sample. Typically, a third-party library comes from an outside source such as a public project or a vendor.

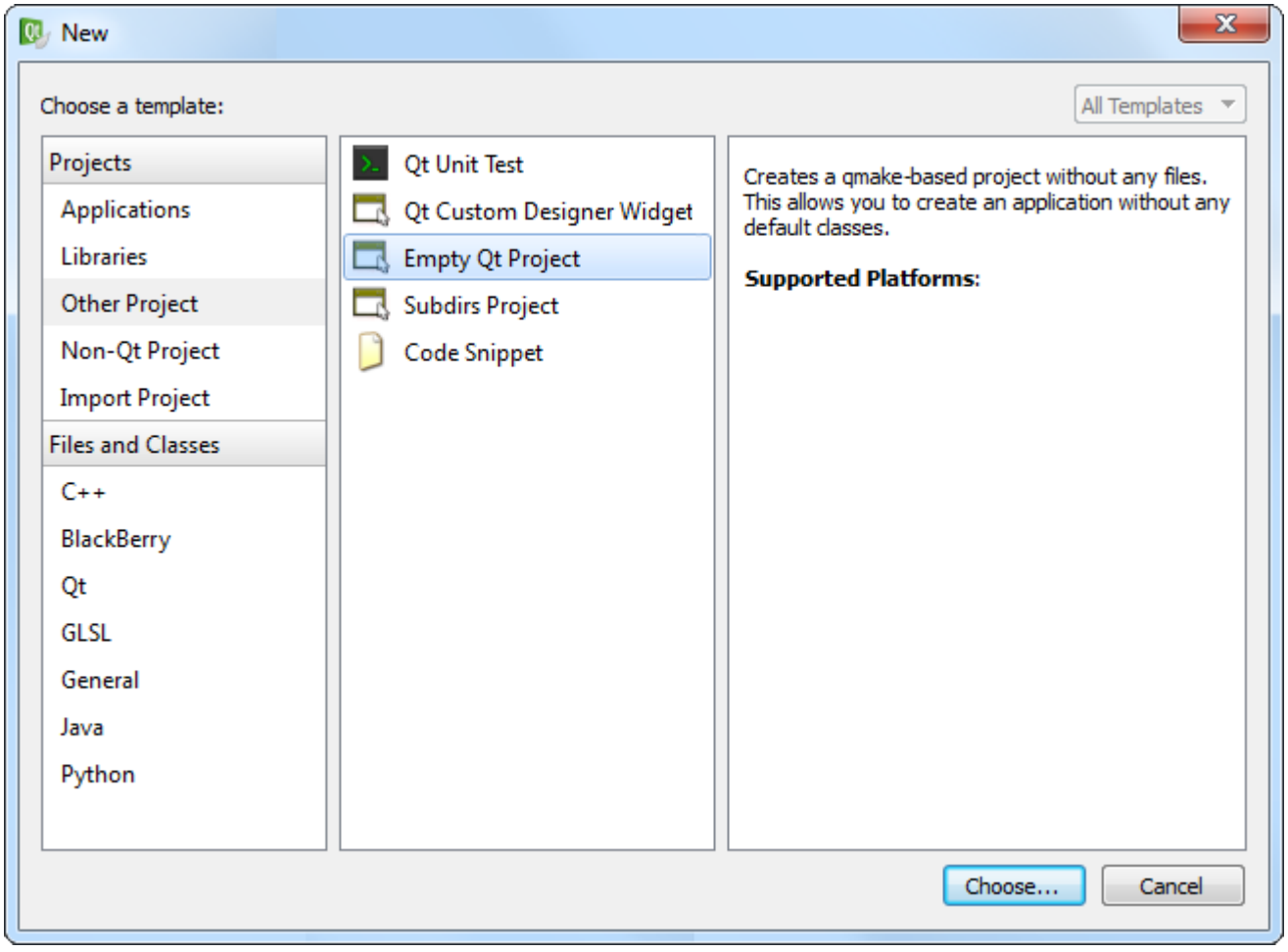
Creating a project for the library

The first stage in generating a library for use by Qt apps is to create a project in Qt Creator and define library functions.

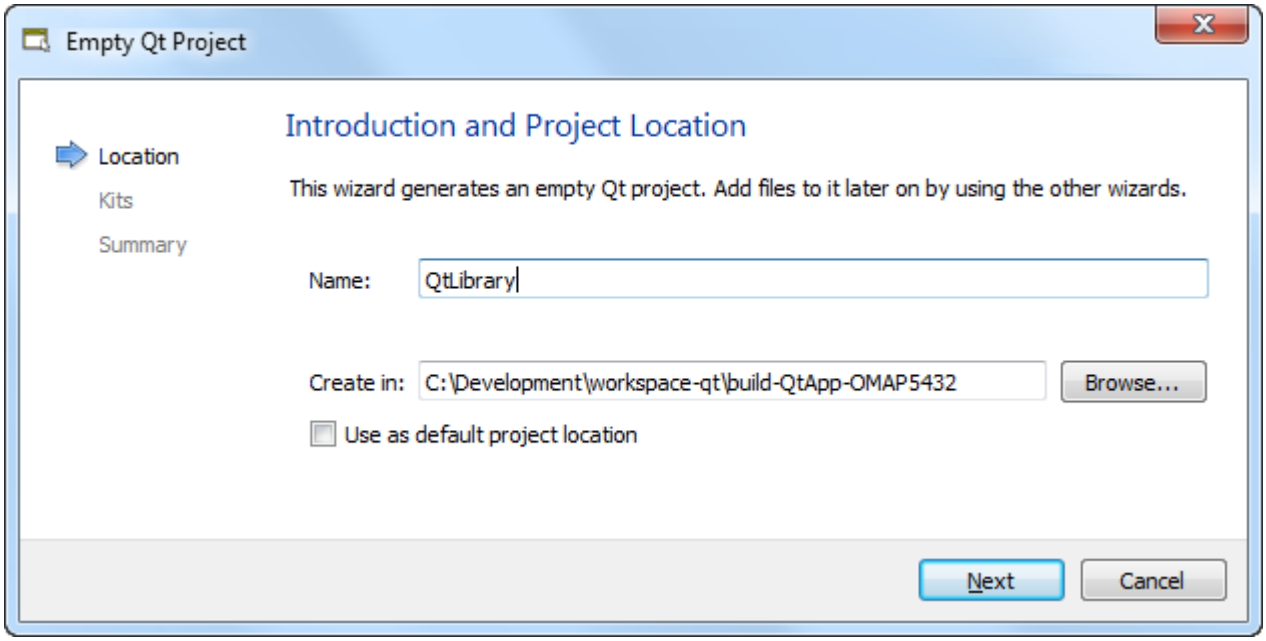
This example builds a Qt project that compiles into a dynamic library (**.so**) file. The library exports a public function that can be called by application code.

To create a Qt project and configure its project file:

1. Launch Qt Creator.
2. In the **File** menu, choose **New File or Project...**
3. In the resulting dialog, choose **Other Project** from the list on the left, then **Empty Qt Project** from the list in the middle, and then click **Choose...**



4. In the **Location** page of the **Empty Qt Project** dialog, name the project `QtLibrary`, then click **Next**.



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**.

   To define a kit, you must first define toolchain settings (e.g., compiler, debugger), as explained in "*Configuring a toolchain in Qt Creator* (p. 20)".

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

   Qt Creator creates the new project and displays the empty **QtLibrary.pro** file in the editing area.

7. Add these lines to this file:

   ```
   # We're building a library
   TEMPLATE = lib
   VERSION = 1.0
   ```

   This instructs Qt Creator to build a dynamic library file with the indicated version number. The resulting file will be called **libQtLibrary.so.1.0**.

The project file can define many variables that affect how **qmake** builds the project; for the full list, see the *Variables | QMake* reference in Digia's online Qt documentation.

Adding a function

After the library project is created, you can add functions to export services to applications.

To add a function:

1. Click the **Edit** icon on the left side, right-click the `QtLibrary` folder in the **Projects** view, then choose **Add New...** in the popup menu.
2. In the **New File** dialog, select `C++` in the **Files and Classes** list, then `C++ Class` in the list of file types (shown in the middle), then click **Choose...**
3. In the **Details** page of the **C++ Class Wizard** dialog, name the class `Foo`, then click **Next**.
4. In the **Summary** page, click **Finish**.

   Qt Creator creates two new files, **foo.h** and **foo.cpp**, and adds them to the project.
5. Open **foo.h** for editing (by double-clicking its entry in the **Project** view), and add this content to the file:

```
#ifndef FOO_H
#define FOO_H

#include <QString>

class Foo
{
public:
    Foo();
    QString message() const;
};

#endif // FOO_H
```

The *message()* function is declared in the public part of the class so it's visible to application code outside of the library.

6. After saving the header file, edit **foo.cpp** to add this content:

```
#include "foo.h"

Foo::Foo()
{
}

QString Foo::message() const
{
    return QStringLiteral("QtLibrary says hello world");
}
```

We define the most basic function that simply returns a string to its caller, just to illustrate the mechanism for implementing library functionality. You'll write functions that do more useful actions but the method of defining them in library projects is always the same.

You can now build the library into an **.so** file containing the defined functionality.

Building the library

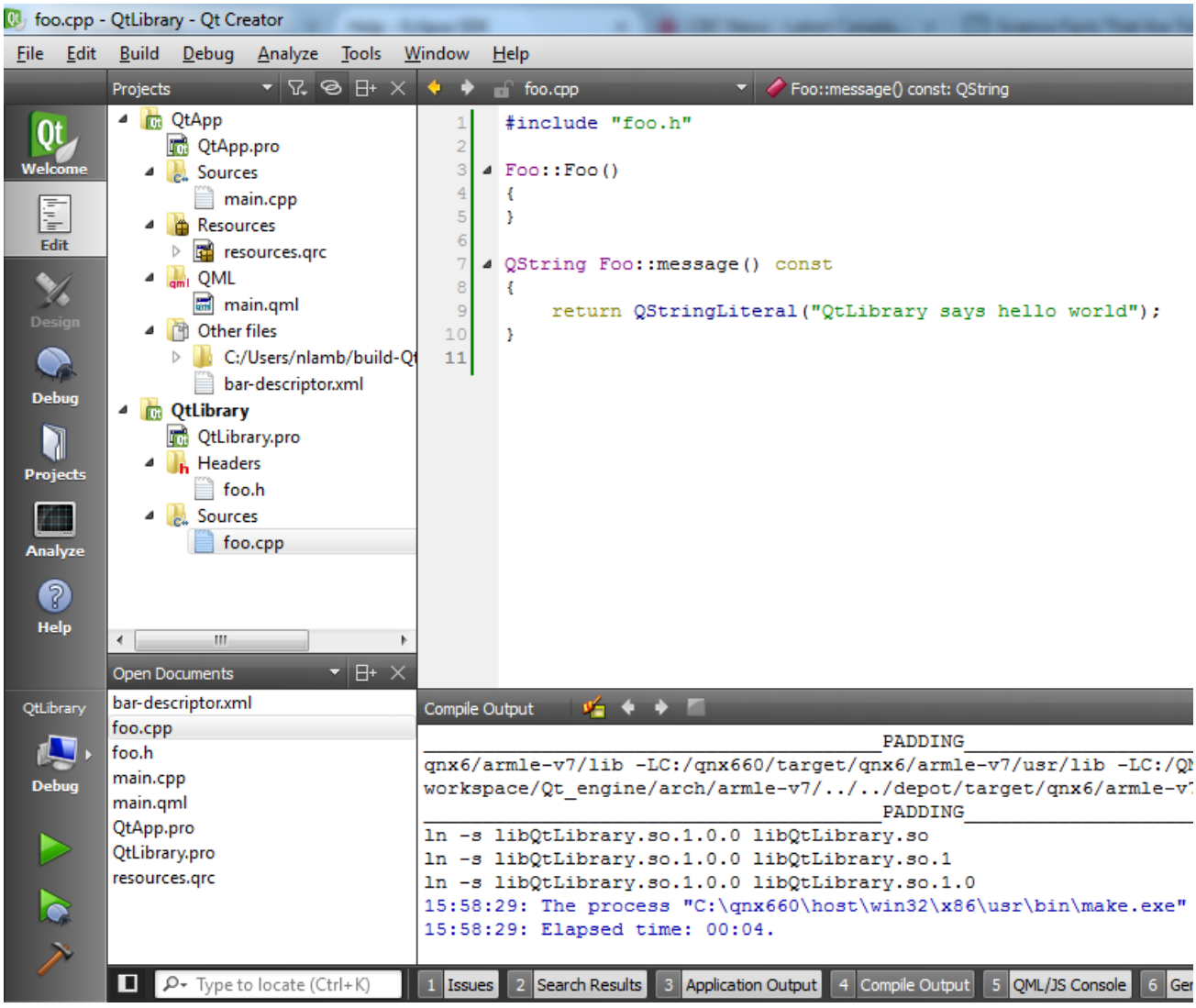
After defining functions for the library, you can build its shared library (**.so**) file so that applications can dynamically link in the library functionality.

Qt Creator has many features to make compilation and debugging easier, as explained in "*Tips for compiling programs in Qt Creator* (p. 42)".

To compile the library:

- Select **Build → Build Project "QtLibrary"**.

  Qt Creator starts building the library and displays the QCC output in the **Compile Output** window.



If the build succeeds, the library file will be in the directory specified in the **General** section of the **Build Settings** page, which is accessed by clicking the **Projects** icon on the left side and then selecting the QtLibrary project.

If the build fails, you can review the messages shown in the **Compile Output** window (which is accessed by clicking the button with the same name at the bottom) to determine the cause of the failure and then take corrective action to fix the project.

Adding the library to Qt app projects

After generating the dynamic library file for **QtLibrary**, you can add the library to the projects of Qt apps to make the library functionality available to those apps.

In this tutorial, we modify the project for QtApp (which we created in *Creating and running Qt apps* (p. 25)) to include our new library.

To add the library to QtApp:

1. Click the **Edit** icon on the left side, right-click the `QtApp` folder in the **Projects** view, then choose **Set "QtApp" as Active Project** in the popup menu.
2. Right-click the `QtApp` folder again, then choose **Add Library...**
3. In the **Type** page of the resulting dialog, select **External Library**, then click **Next**.
4. On the **Library file** line in the **Details** page, click the **Browse** button (shown on the right) to open the file selector.
5. Navigate to the build directory of QtLibrary, select **libQtLibrary.so**, then click **Open**.
6. On the **Include path** line, click **Browse** to open the file selector.
7. Navigate to and select the source directory of QtLibrary, then click **Open**.
8. Under the **Platform** heading, uncheck the boxes for `Mac` and `Windows`, then click **Next**.

   This last step is necessary because QNX is a POSIX-compliant OS so it uses the Linux linking convention.
9. On the **Summary** page, click **Finish**.

The **QtLibrary** library is now part of QtApp, meaning the library's functions can be called from QtApp code.

Calling library functions in Qt apps

With **QtLibrary** integrated with QtApp, you can now write code that uses the library's *message()* function.

To call a library method in QtApp code:

1. Open **main.cpp** for editing (by double-clicking its entry under **QtApp** in the **Project** view), and replace its contents with this code:

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

#include "foo.h"

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

    // Get the screens so we can dynamically size our display
    QScreen* screen = QGuiApplication::primaryScreen();

    // Quit if there's no screen connected
    if (screen == NULL) {
        return 1;
    }

    // Get the width and height of the display
    int w = screen->size().width();
    int h = screen->size().height();

    QQuickView view;
    Foo foo;
    QString msg = foo.message();
    view.rootContext()->setContextProperty("_message", msg);

    // Set up the view to have the proper size
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.resize(w, h);

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

    return app.exec();
}
```

The code in **main.cpp** uses the library by creating a **Foo** object, calling the object's *message()* function, and then making the returned string available to QML so it can be displayed.

2. Open **main.qml** for editing and replace its contents with this code:

```
import QtQuick 2.0

Rectangle {

    Text {
        text: _message
        anchors.centerIn: parent
    }
}
```

3. Build the app by selecting **Build → Build Project "QtApp"**.

The app is built with integrated QtLibrary functionality and can run on the target, so long as the library file is packaged with it.

Packaging Qt apps with the library

After the library has been added to the project of a Qt app, the packaging process for the app remains mostly the same, except for two extra steps.

To package QtApp so it can use QtLibrary, you must:

1. Edit the project file (**QtApp.pro**) to add this line:

   ```
   QMAKE_LFLAGS += "-Wl,-rpath,app/native/lib"
   ```

   This instruction embeds the path of the library file (**libQtLibrary.so**) into the QtApp binary. When the **launcher** service runs QtApp in its sandbox environment, the service uses a root path of **app/native**. All files within the Blackberry ARchive (BAR) package are relative to this location. For instance, from the perspective of QtApp, its icon file is found at **app/native/icon.png**.

   To package **libQtLibrary.so** into the **lib** subdirectory in the BAR file, we set `rpath` to the root path appended with this subdirectory (i.e., **app/native/lib**).

2. Update the arguments for the packaging command as follows:

   ```
   QtApp.bar %{sourceDir}\bar-descriptor.xml QtApp
    -C %{sourceDir} %{sourceDir}\icon.png
    -e ProjectBuildDir\libQtLibrary.so.1.0 lib/libQtLibrary.so.1
   ```

   The newly added `-e` option is followed by two paths. The first is the library file's build location on the host system (in this case, replace *ProjectBuildDir* with the path containing the output library file) and the second is the relative location of the library file within the BAR package. Note that the file is purposely renamed from **libQtLibrary.so.1.0** to **libQtLibrary.so.1**.

   The directory separators in this example are backslashes (`\`), which are used for Windows, but you must use the appropriate separator for your OS (i.e., `"/"` if you're running Linux). The exception is the second path for `-e`; this must use the Linux separator because it specifies a relative location on the QNX target, which follows the POSIX directory convention.

   If you're using Qt Creator to package the app, you must access the **Build & Run** tab and edit the build step for the packaging command to add these arguments, as explained in "*Packaging the app into a BAR file from Qt Creator* (p. 44)". In the above example, *ProjectBuildDir* is the build directory specified in the **General** section of the **Build Settings** page.

   You can also package the app from the command line, by passing these arguments to **blackberry-nativepackager** in a BlackBerry 10 OS terminal, as described in "*Packaging the BAR file from the command line* (p. 47)".

Chapter 5
Writing an HMI

You can develop your own HMI for QNX Apps and Media targets using Qt Creator. The process for writing an HMI is similar to that of writing Qt apps except for the packaging (because the HMI is a standalone application and not packaged as a BAR file).

The sections that follow provide a walkthrough of writing an HMI. The major steps include:

1. Defining the project components (e.g., resource file, main UI file, C++ entry point file).
2. Compiling, running, and debugging the HMI application on a target system.
3. Adding controls for various subsystems (e.g., volume) to expand the HMI capabilities as needed.

To develop a Qt-based HMI, you must have the necessary Qt tools installed and configured on your host system, as explained in *Preparing your host system for Qt development* (p. 13).

Creating a project for a Qt HMI

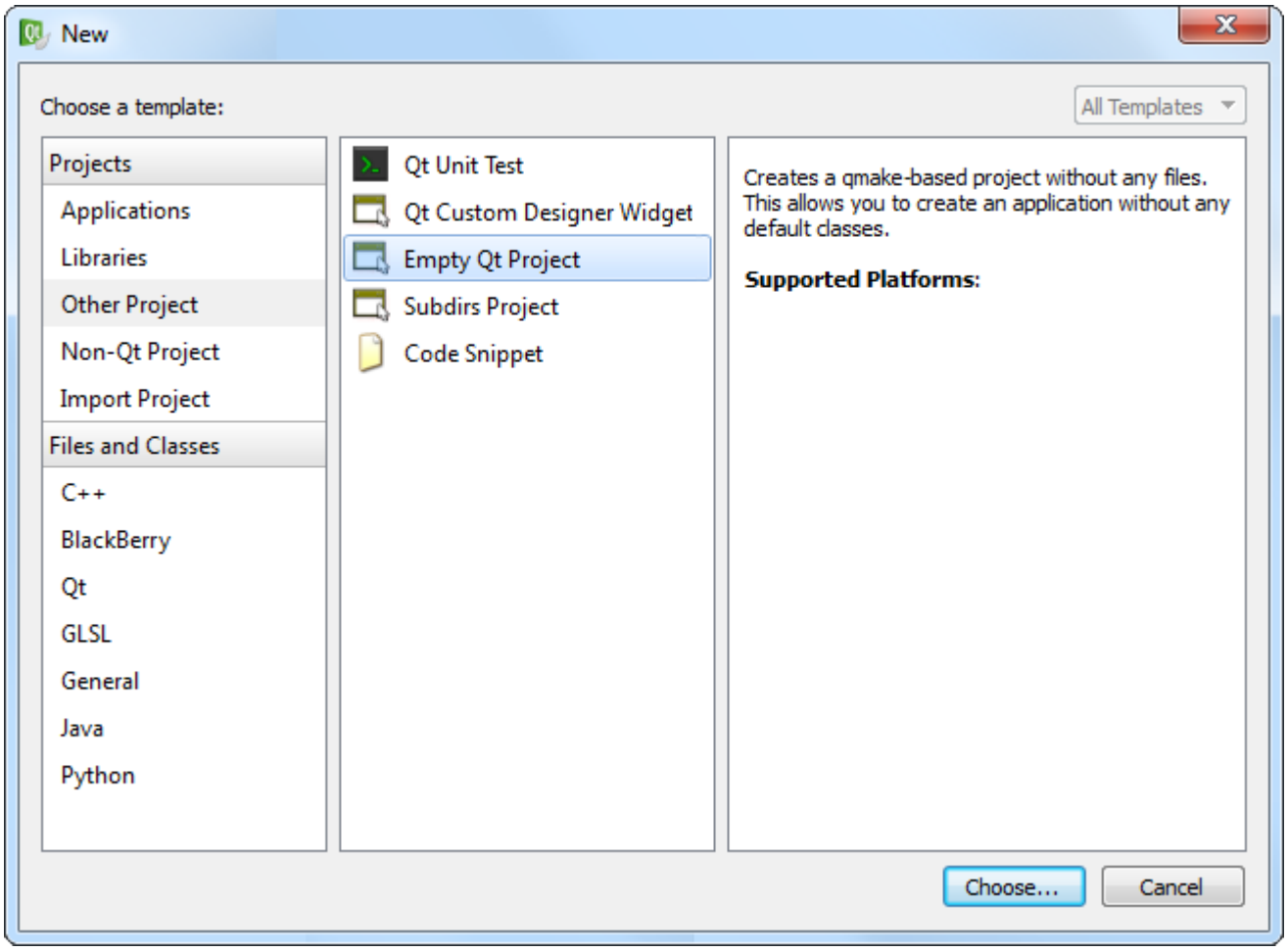
The first stage in writing a Qt HMI is to create a project in Qt Creator and add the files that define the UI, application entry point, and how to package the project components.

In particular, the project will contain:

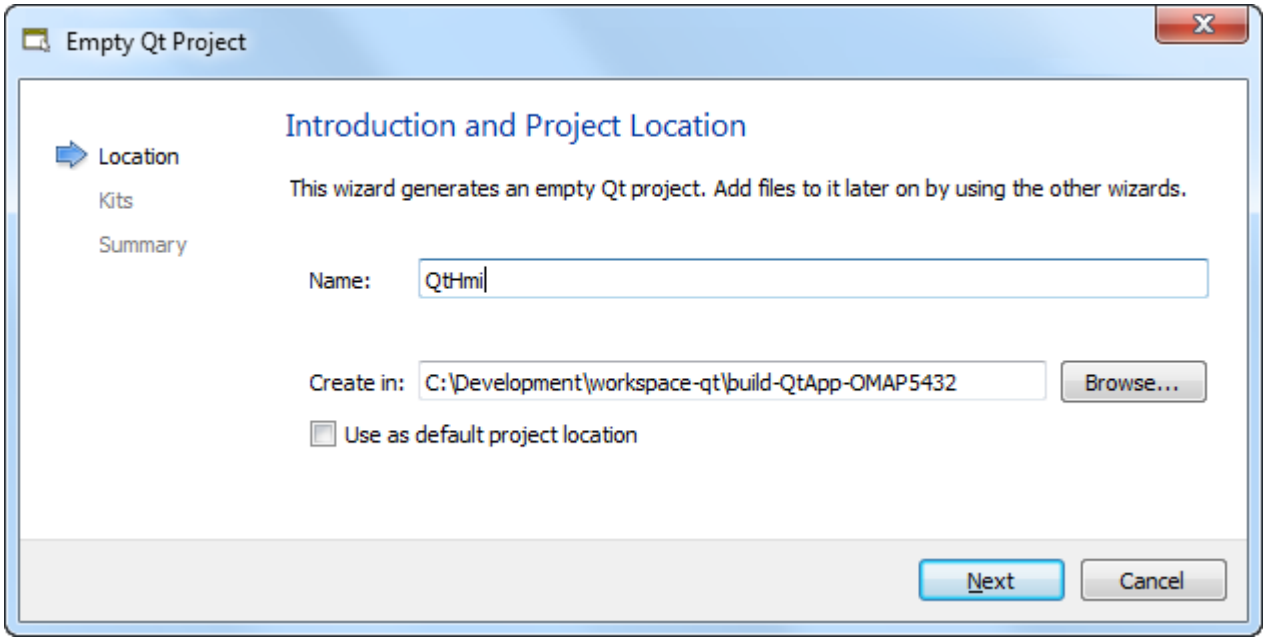
- A Qt Project file (**QtHmi.pro**) to store the project configuration settings
- A QML file (**main.qml**) to define the main UI elements for the application
- A QRC file (**resources.qrc**) to package the project resources into the binary
- A CPP file (**main.cpp**) to contain the entry-point function for starting the application

To create a Qt project and start defining its project file:

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



4. In the **Location** page of the **Empty Qt Project** dialog, name the project `QtHmi`, then click **Next**.



All files related to the project—C++ and QML source code, resource files, and the project configuration file—will be stored in the folder specified on the **Create in** line in this dialog.

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**.

   To define a kit, you must first define toolchain settings (e.g., compiler, debugger), as explained in "*Configuring a toolchain in Qt Creator* (p. 20)".

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

   Qt Creator creates the new project and displays the empty **QtHmi.pro** file in the editing area.

7. Add the following lines to this file:

   ```
   # We're building an app
   TEMPLATE = app

   # This is the name to give the compiled application
   TARGET = QtHmi
   ```

   This action configures the project to build an application binary (as opposed to a dynamic or static library).

The project file can define many variables that affect how **qmake** builds the project; for the full list, see the *`Variables | QMake`* reference in Digia's online Qt documentation.

Adding the main QML file

Next, you can add a QML file to define the UI for the application.

To define the main QML file for your HMI:

1. Click the **Edit** icon on the left side, right-click the `QtHmi` folder in the **Projects** view, then choose **Add New...** in the popup menu.
2. In the **New File** dialog, select `Qt` in the **Files and Classes** list, then `QML File (Qt Quick 2)` in the list of file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the **New QML file** dialog, name the file `main`, then click **Next**.
4. In the **Summary** page, click **Finish**.

   Qt Creator adds **main.qml** to the project and opens this file in the editing area.

5. Replace the contents of this file with the following:

```
import QtQuick 2.0

Rectangle {

    color: "black"

    Text {
        color: "white"
        text: qsTr("Awesome HMI goes here")
        anchors.centerIn: parent
    }
}
```

6. After saving the QML file, edit the **QtHmi.pro** file to add the following lines:

```
# The Qt modules needed for this project
QT += quick
```

   This informs Qt Creator that the project uses the `quick` module, which is needed to build QML-based UIs.

Adding the QRC file

To make it easier to deploy and run the application on the target, you can include the main QML file in a Qt resource (QRC) file. A resource file packages many components including QML files, images, and fonts into the binary so you don't have to deploy them alongside the binary on the target.

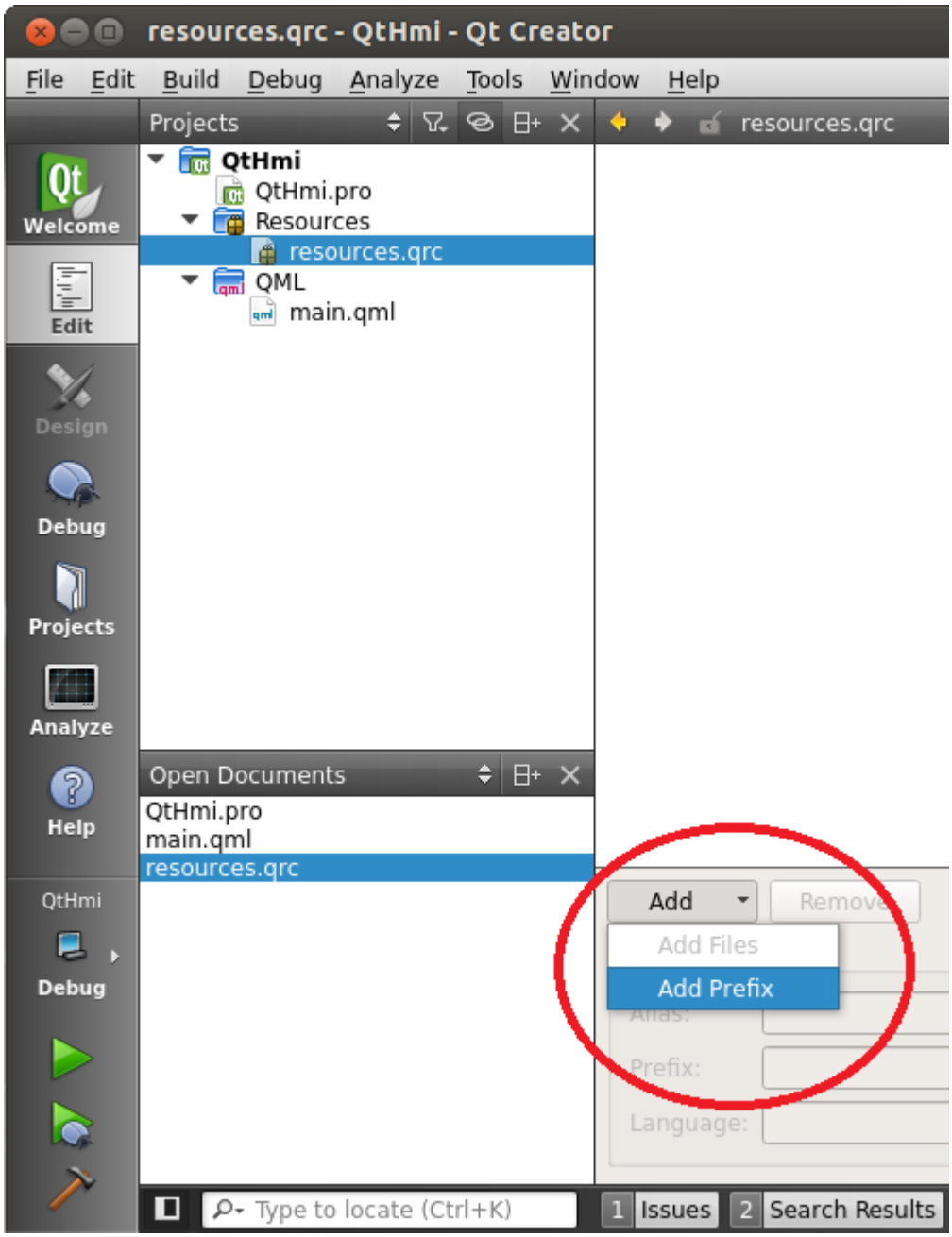
In addition to compiling resources into their binaries, applications can access resources directly from the target's filesystem. Deciding whether to use a resource file is a design decision. More information about resource files and how to package Qt binaries can be found on Digia's Qt website: ***http://qt-project.org/doc/qt-5/resources.html***.

To add a QRC file and include the main QML file in it:

1. In the **Project** view, right-click the `QtHmi` folder and click **Add New...**
2. In the **New File** dialog, select `Qt` in the **Files and Classes** list, then `Qt Resource file` in the list of file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the resulting dialog, 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 and opened in Qt Creator for editing.

5. In the configuration area near the bottom, click **Add**, then choose **Add Prefix**.



6. In the **Prefix** field, enter `qml`.

   Prefixes add structure to the resource file. Any prefix scheme can be used, as long as you organize your resources in a way that makes sense for the developers working on the project.

7. Click the **Add** button again, then choose **Add Files**.
8. In the file selector that the IDE opens, navigate to and select **main.qml**, then click **Open**.

   This QML file is found in the folder specified on the **Create in** line in the **Empty Qt Project** dialog, which was opened when the project was created.

The QML file is now part of the Qt resource file that will be compiled into the binary.

Adding the CPP file

The last step in creating a project for an HMI is to add the C++ code that runs the application and loads the QML file.

To add a CPP file that starts the application:

1. In the **Project** view, right-click the `QtHmi` 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 file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the **New C++ Source File** dialog, name the file `main`, then click **Next**.
4. In the **Summary** page, click **Finish**.

   A new file, `main.cpp`, has been added to the project and opened for editing.

5. Add the following code to this file:

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

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

    // Get the screens so we can dynamically size our display
    QList<QScreen*> screens = QGuiApplication::screens();

    // Quit if no screen is connected
    if (screens.empty()) {
        return 1;
    }

    // Get the width and height of the display
    int w = screens[0]->size().width();
    int h = screens[0]->size().height();
```

```
    QQuickView view;

    // Set the main QML user interface file to this view
    view.setSource(QUrl("qrc:/qml/main.qml"));

    // Set up the view to have the proper size
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.resize(w, h);

    // Show our user interface
    view.show();

    return app.exec();
}
```

Note that the `view.setSource()` call uses the `qrc:` prefix for the **QUrl** object. This is how the application accesses resources in the **resources.qrc** file.

You now have a shell Qt application ready to go!

Building the HMI application for a QNX target

After creating a Qt project and defining the necessary resources for the HMI, you can build the application binary and then deploy and run it on the target.

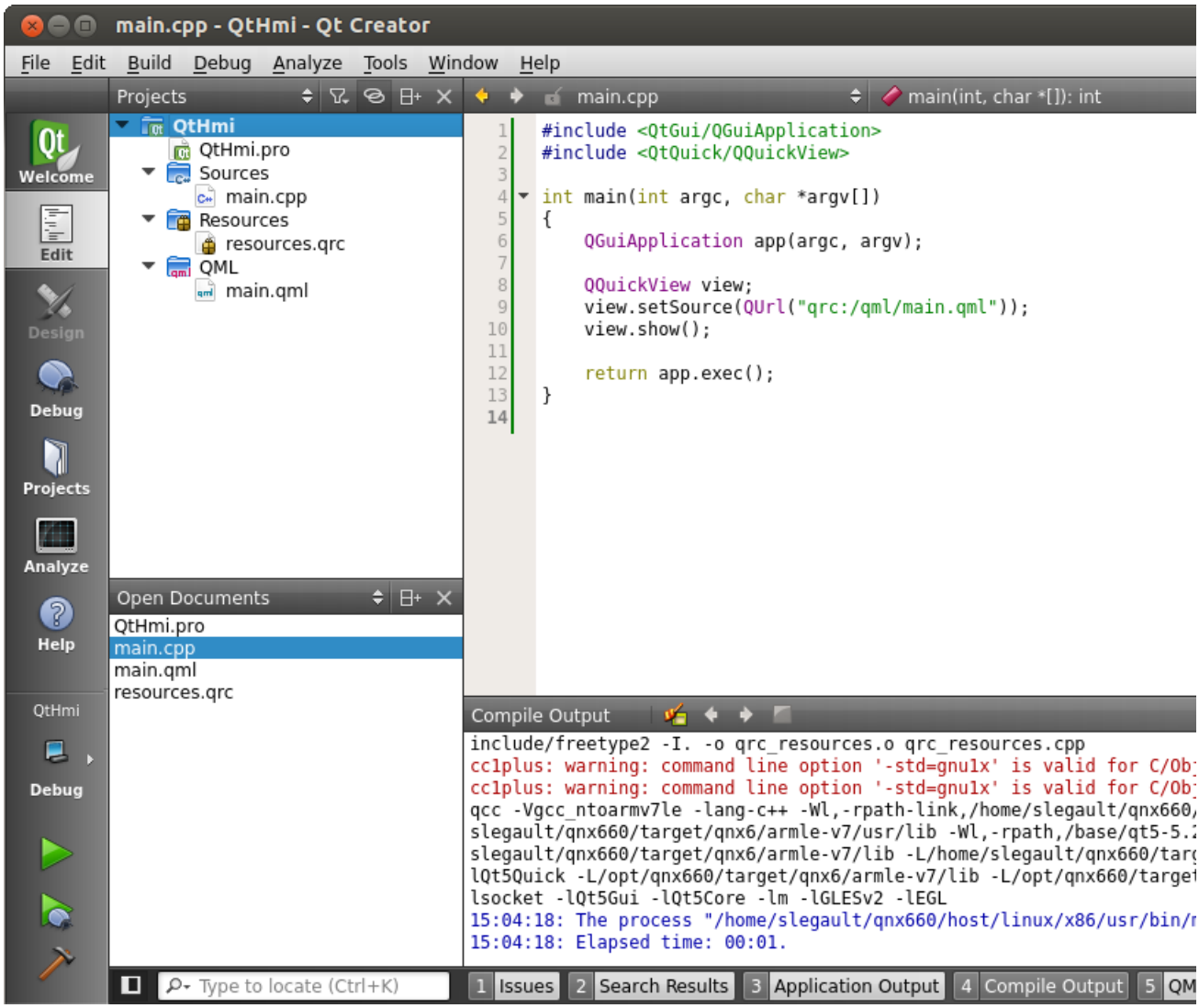
To build and run HMI applications written for QNX Apps and Media 1.0, follow these same steps.

To compile the HMI application:

- In the **Build** menu, choose **Build Project "QtHmi"**.

  If you're rebuilding a legacy application, your project will likely be named something other than **"QtHmi"**.

  Qt Creator starts building the application and displays the QCC output in the **Compile Output** window.



If the application builds successfully, the binary will be in the build directory specified in the **General** section of the **Build Settings** page, which is accessed by clicking the **Projects** icon on the left side and then selecting the project for the HMI that you want to run.

If the build fails, you can review the messages shown in the **Compile Output** window (which is accessed by clicking the button with the same name at the bottom) to determine the cause of the failure, and then fix the project as necessary.

Qt Creator has many features to make compilation and debugging easier, as explained in "*Tips for compiling programs in Qt Creator* (p. 42)".

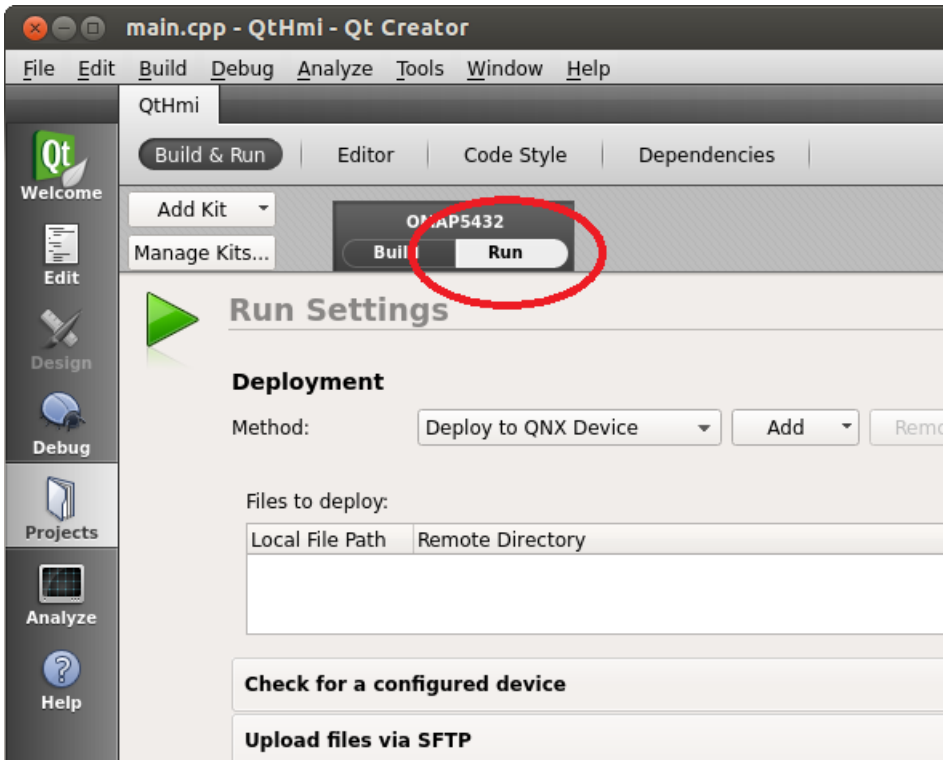
Configuring the runtime environment

Before running the HMI on the target, we recommend setting the QQNX_PHYSICAL_SCREEN_SIZE environment variable. This variable defines the application display dimensions, to ensure that the HMI fits the target's display.

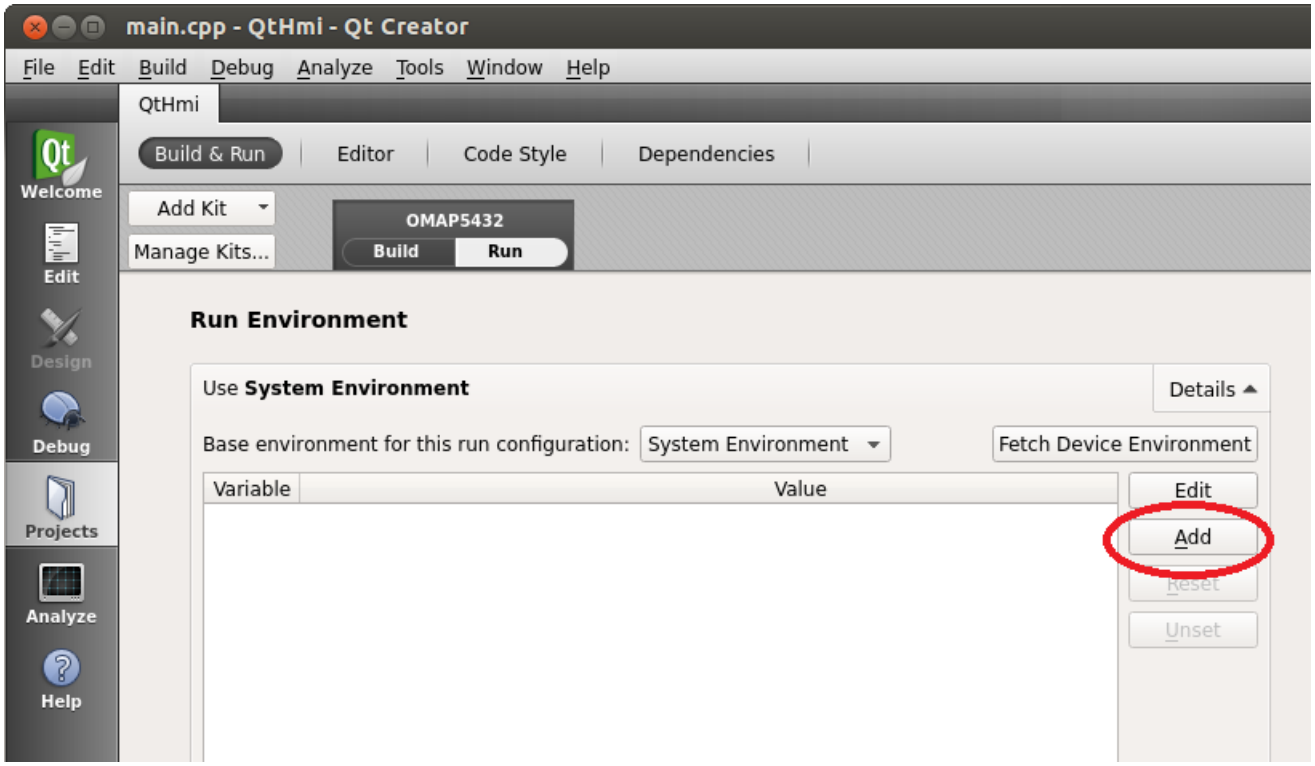
CAUTION: If this variable isn't set, the application will still run but you'll receive an **stdout** warning and the application might not display correctly; see "*Environment Variables* (p. 34)" for more information.

To configure the runtime environment:

1. Click the **Project** icon on the left side to access the **Build & Run** settings.
2. Click the **Run** tab to switch to the **Run Settings** page.



3. Scroll down to find the **Run Environment** heading, then expand the **Use System Environment** entry.
4. Click the **Add** button on the right side to add an environment variable.



5. Set the variable name to QQNX_PHYSICAL_SCREEN_SIZE and the value to the display dimensions, in millimeters, of your target.

   The value you specify must contain the display width and height, separated by a comma. For example, when using a 150 mm by 90 mm display, enter `150,90`.

The target runtime environment is now configured to display the HMI.

Uploading the binary to the target

You can specify the target path for installing the HMI binary and upload the binary from Qt Creator.

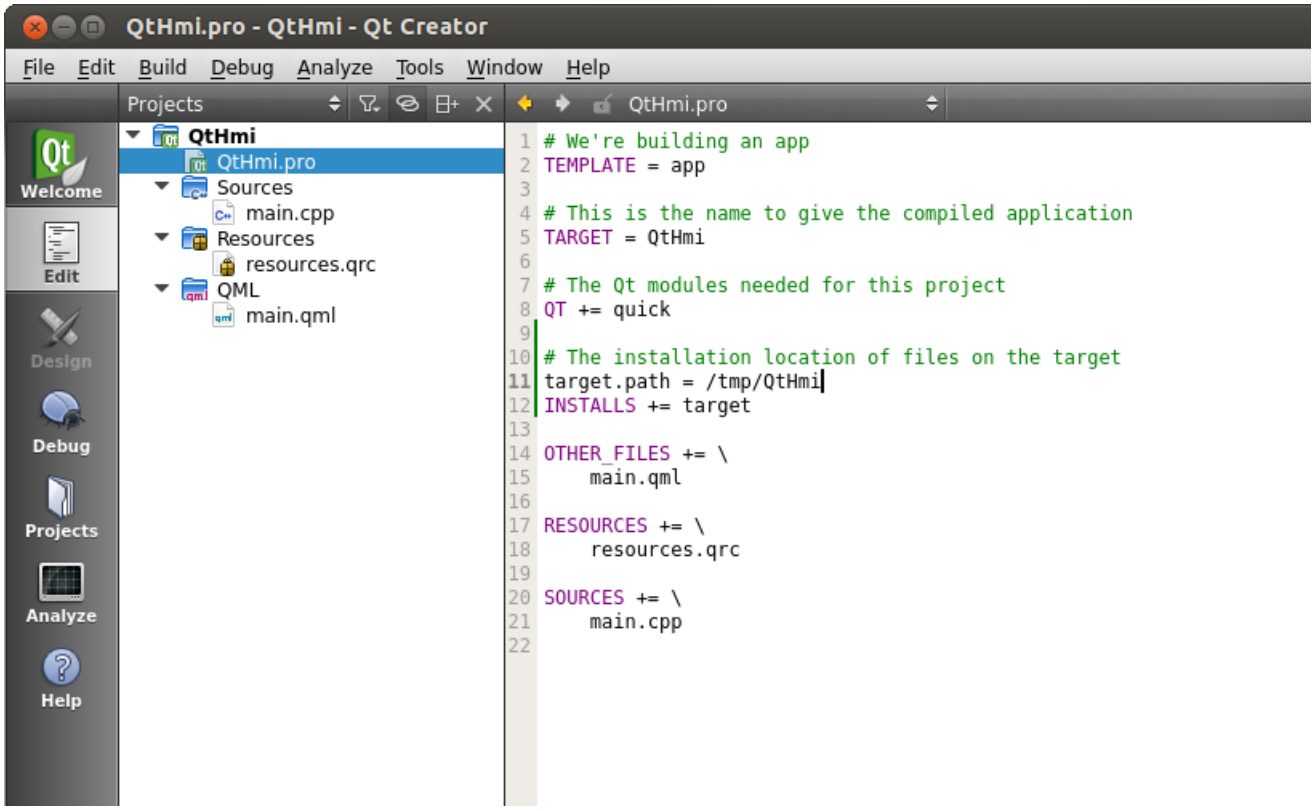
To define the target path and upload the binary:

1. Edit the **QtHmi.pro** file to add the following lines:

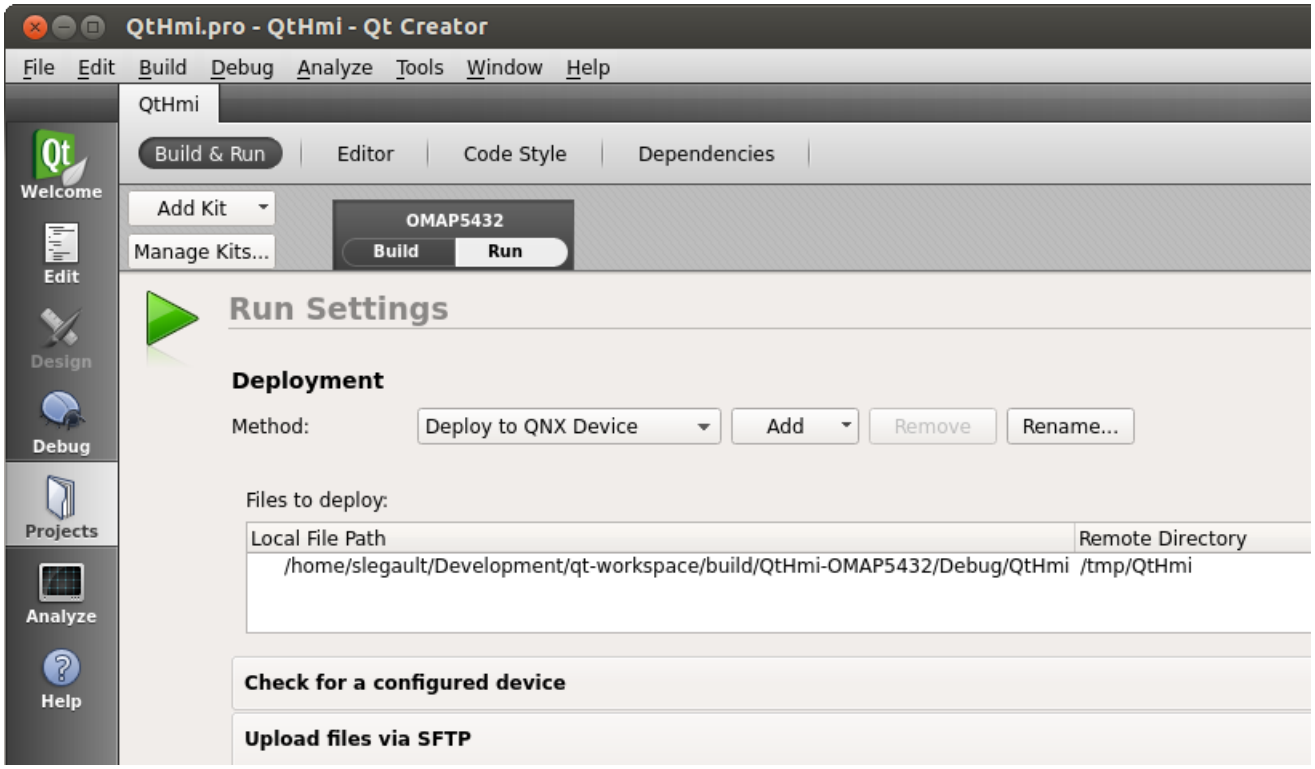
```
# The installation location of files on the target
target.path = /tmp/QtHmi
INSTALLS += target
```

   These lines tell Qt Creator where to upload files on the target.

   At this point, the project file should look like this:



2. Verify the upload location by:
   a) Switching to the **Project** tab by clicking its icon on the left side.
   b) Selecting the **Run Settings** page by clicking its tab towards the top of the **Build & Run** display.
   c) Under the **Deployment** section, the **Files to deploy** box should have an entry that lists the correct build path on the host and **/tmp/QtHmi** as the remote directory.



3. Switch back to the **Edit** tab. From the menu bar, choose **Build → Deploy Project "QtHmi"**. This uploads the binary to the target.

Running the application from Qt Creator will automatically deploy the binary if it has changed since the last deployment.

Running the HMI application

You can now run your HMI on the target using Qt Creator.

To run the HMI application:

1. If a QNX Apps and Media image is running on the target, establish an SSH connection with the target and enter the following command to stop the default HMI:

   ```
   # slay -12 homescreen
   ```

   Root permission is required to slay the homescreen process.

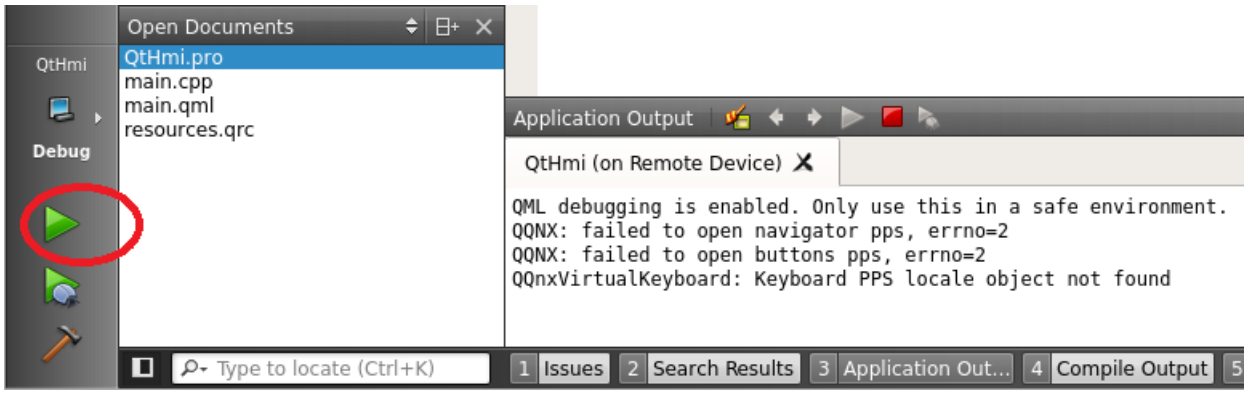
   You have to stop the default HMI to ensure that your new HMI appears on the screen. The default HMI runs in the foreground and any application that you launch will have a z-order less than that of the default HMI and hence, won't be visible (and no error message will be displayed).

   You must slay the homescreen also when you want to run an application developed with QNX SDK for Apps and Media 1.0 on a target running QNX Apps and Media 1.1. This is because when Qt

Creator runs an application, it simply copies it to the target and executes it (without considering z-order).

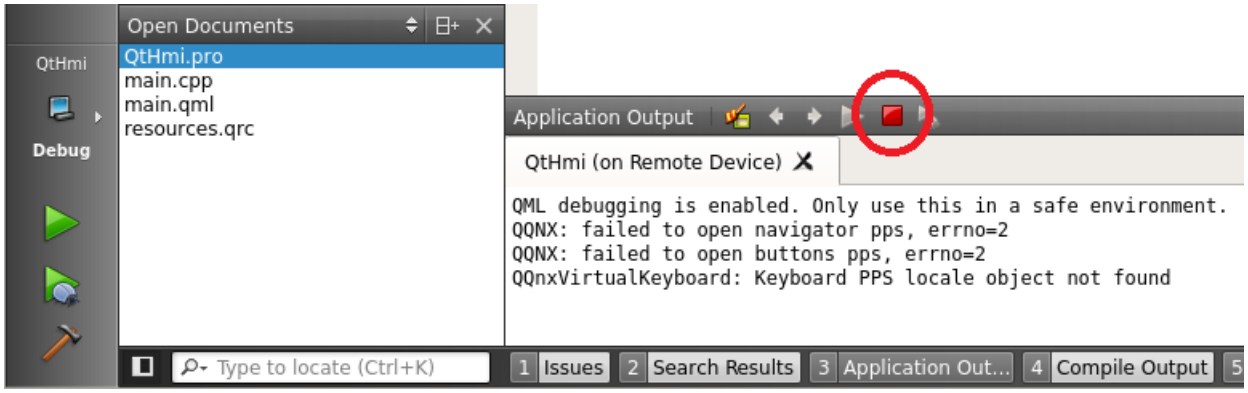
Slaying the homescreen makes the new HMI visible but you may encounter other problems due to the *Application and Window Management* components that are still running. If you don't intend to run packaged apps on your target, a better long-term solution is to disable these components and the homescreen by reconfiguring **/var/etc/services-enabled**. Instructions on doing this are given in the "Full Screen HMI" section of the *User's Guide*.

2. To run the application, click the green Run button in the bottom left corner.



The HMI application runs and you should see it on the screen of the target:

3. To stop the application, click the red Stop button along the top of the **Application Output** window at the bottom.



Qt Creator stops the application and displays a message saying the application was user-terminated and containing the exit code, in the **Application Output** window.

Adding a control to the HMI

Getting the HMI application to run on the target and appear as expected on the screen is an essential step in HMI development. You can then extend the HMI by adding controls to specific services in your embedded system.

We will write a control for setting the audio volume. Specifically, we will define a UI component (using QML) to provide volume adjustment controls and also write the QPPS library calls (using C++) to publish the latest volume level to the audio status PPS object.

Like the Home screen app included with the reference image, our sample control updates the audio status to reflect the latest volume setting but it doesn't actually change the volume of the audio output. This last task involves sending commands to the Audio Manager service through PPS and is beyond the scope of this HMI-writing tutorial.

Summary of steps

In this section of the tutorial, we will:

1. Add the source code for the QPPS library to the HMI project, to build our own copy of the library.
2. Define a new C++ class (**VolumeModule**) to act as the interface between the QML code and the **QPPS** classes.
3. Add image resources for the HMI volume control.
4. Define new QML components (**VolumeUI** and **VolumeSlider**) to create the UI for the audio volume control.

Compiling the QPPS library code with the application

To use QPPS classes to access PPS objects, we copy the QPPS source code into the QtHmi folder and then build the QPPS library functionality into the application.

The QPPS 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. The source code for this library is included in the Qt source code package that's part of the platform installers.

To compile the QPPS library code into QtHmi:

If you have already unpackaged the Qt source code and remember the location where you stored the QPPS library files, you can skip to Step *3* (p. 86).

1. Access the Qt source code package and locate the QPPS library code.

   By default, the installers copy the package to ***DEFAULT_SDP_PATH*/source/appsmedia_1_1_qt_source.zip**. Within the package, the QPPS library code is found at this path: **/qt/src/Homescreen/qpps/**. This last directory contains another directory named **qpps**, which stores the actual source code.

2. Unzip the contents of the **/qt/src/Homescreen/qpps/** directory (including the nested **qpps** directory) to your project directory (e.g., **C:\users*username*\QtHmi**).

The **qpps** subdirectory is added to your project directory and contains the header and class definition files needed to use QPPS classes in the QtHmi code.

3. In Qt Creator, click the **Edit** icon on the left side, right-click the `QtHmi` folder in the **Projects** view, then choose **Add Existing Directory...**
4. In the resulting dialog, on the **Source directory** line, click **Browse** to open the file selector.
5. Navigate to the directory containing the QPPS library code, then click **Select Folder**.
   The newly selected directory is listed on the **Source directory** line.
6. Click **OK** to close the dialog.

The **QPPS** header and class definition files have been added to the QtHmi project, giving you access to the QPPS classes. When you build the HMI application, the library functionality will be built into the binary, ensuring that the application runs whether or not the target contains the QPPS library file.

In addition, Qt Creator has added this content to the project file (**QtHmi.pro**):

```
SOURCES += \
    main.cpp \
    qpps/dirwatcher.cpp \
    qpps/object.cpp \
    qpps/variant.cpp

HEADERS += \
    qpps/changeset.h \
    qpps/dirwatcher.h \
    qpps/dirwatcher_p.h \
    qpps/object.h \
    qpps/object_p.h \
    qpps/qpps_export.h \
    qpps/variant.h
```

Adding the VolumeModule C++ class

The **VolumeModule** class acts as the interface between the QML-based UI and the QPPS library. This C++ class exposes the volume level, which is read through PPS, as a **Q_PROPERTY** consumable by QML.

To add the **VolumeModule** class:

1. In the **Projects** view, right-click the `QtHmi` folder, then choose **Add New...**.
2. In the **New File** dialog, select `C++` in the **Files and Classes** list, then `C++ Class` in the list of file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the resulting dialog, name the file `VolumeModule`, then click **Next**.
4. In the **Summary** page, click **Finish**.
   Two new files are added to the project (**volumemodule.h** and **volumemodule.cpp**).

5. Edit **volumemodule.h** and replace the contents with the following code:

```
#ifndef VOLUMEMODULE_H
#define VOLUMEMODULE_H

#include <QObject>
#include "qpps/object.h"

class VolumeModule : public QObject
{
    Q_OBJECT

    // Volume setting, accessible by QML
    Q_PROPERTY(double volume READ volume WRITE setVolume
               NOTIFY volumeChanged)

public:
    // Constructor
    explicit VolumeModule(QObject *parent = NULL);

    Q_INVOKABLE double volume() const;

    Q_INVOKABLE void setVolume(const double value) const;

public Q_SLOTS:
    // Updates volume level when volume change is reported by PPS
    void audioStatusChanged(const QString &name,
                            const QPps::Variant &attribute);

Q_SIGNALS:
    // Emitted when the volume level changes
    void volumeChanged();

private:
    // Reference to PPS object containing audio volume level
    QPps::Object *m_ppsAudioStatus;

    // Volume setting
    double m_volume;
};

#endif // VOLUMEMODULE_H
```

In this code excerpt, the include path for the header file that defines the **QObject** class is a relative path (**qpps/object.h**). This is because in our example, we copied the QPPS header files to the **qpps** subdirectory within the project directory. If you unpackaged the QPPS library code to a different location, you must adjust the include path accordingly.

6. Edit **volumemodule.cpp** and replace the contents with the following code:

```
#include "volumemodule.h"

VolumeModule::VolumeModule(QObject *parent)
    : QObject(parent)
{
    // Access PPS object that stores audio device status
    m_ppsAudioStatus = new QPps::Object(
        QStringLiteral("/pps/services/audio/status"),
        QPps::Object::PublishAndSubscribeMode, false, this);

    if (!m_ppsAudioStatus->isValid()) {
        // Print error message if unable to read audio device
        // status through PPS
        qCritical("%s Could not open %s: %s", Q_FUNC_INFO,
                qPrintable(m_ppsAudioStatus->path()),
                qPrintable(m_ppsAudioStatus->errorString()));
    }
    else {
        // Connect signal for changed attribute in PPS object
        // to handler for audio status changes
        connect(m_ppsAudioStatus,
            SIGNAL(attributeChanged(QString,QPps::Variant)),
            this,
            SLOT(audioStatusChanged(QString,QPps::Variant)));
    }
}

double VolumeModule::volume() const {
    return m_volume;
}

void VolumeModule::setVolume(const double value) const {
    if (value == m_volume) {
        //Don't set the volume if it's already set to that
        return;
    }
    if (!m_ppsAudioStatus->isValid()) {
        qCritical("%s Could not write %s: %s", Q_FUNC_INFO,
                qPrintable(m_ppsAudioStatus->path()),
                qPrintable(m_ppsAudioStatus->errorString()));
        return;
    }
    if (!m_ppsAudioStatus->setAttribute(
            "output.speaker.volume", value)) {
        qWarning("%s SetAttribute failed %s: %s", Q_FUNC_INFO,
```

```
                qPrintable(m_ppsAudioStatus->path()),
                qPrintable(m_ppsAudioStatus->errorString()));
    }
}

void VolumeModule::audioStatusChanged(
                        const QString &name,
                        const QPps::Variant &attribute)
{
    if (name == QStringLiteral("output.speaker.volume")) {
        m_volume = attribute.toDouble();
        emit volumeChanged();
    }
}
```

7. Edit **main.cpp** to contain the following code:

In the excerpt below, the sections of new code are indicated by comments containing the words `NEW CODE`.

```
#include <QtGui/QGuiApplication>
#include <QtQuick/QQuickView>
#include <QScreen>
#include <QQmlContext>
#include <qqml.h>

// BEGIN NEW CODE
#include "volumemodule.h"

void setupVolumeModule(QQuickView* view)
{
    // Register with the Qt Metatype system
    qmlRegisterUncreatableType<VolumeModule>(
                    "com.mycompany.hmi",
                    1, 0, "VolumeModule",
                    QStringLiteral("Access to object"));

    // By passing in the view as a parent object, the
    // VolumeModule will be deleted when its parent is deleted
    VolumeModule* volumeModule = new VolumeModule(view);

    // Give the view access to the VolumeModule
    view->rootContext()->setContextProperty(
                            QStringLiteral("_volumeModule"),
                            volumeModule);
}
```

```
// END NEW CODE

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

    // Get the screens so we can dynamically size our display
    QList<QScreen*> screens = QGuiApplication::screens();

    // Quit if no screen is connected
    if (screens.empty()) {
        return 1;
    }

    // Get the width and height of the display
    int w = screens[0]->size().width();
    int h = screens[0]->size().height();

    QQuickView view;

    // BEGIN NEW CODE
    // Set up the volume control
    setupVolumeModule(&view);
    // END NEW CODE

    // Set the main QML UI file to this view
    view.setSource(QUrl("qrc:/qml/main.qml"));

    // Set up the view to have the proper size
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.resize(w, h);

    // Show our user interface
    view.show();

    return app.exec();
}
```

This code gives the application access (through the view) to the **VolumeModule** class. Although this class isn't coded as a singleton at the Qt level, from the QML layer, the class is accessed by a singleton object called `_volumeModule`.

The C++ code needed for the volume control is complete. Next, you can define the UI components that allow the user to adjust the volume.

Adding images for volume control

You can copy the images related to volume control shown here to your host system and then add them as project resources so your HMI can display them.

To add images for volume control to your HMI project:

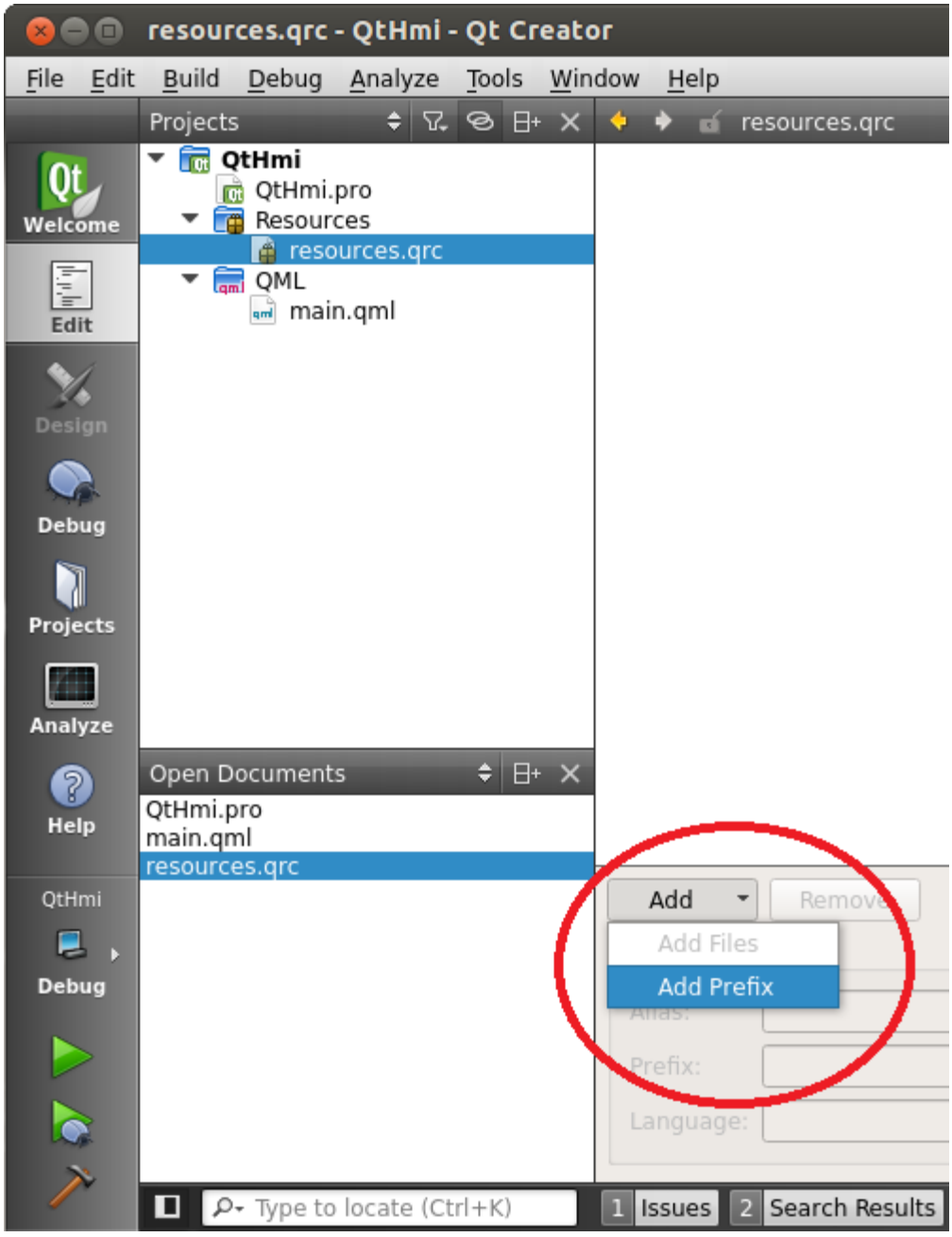
1. Copy these images to your project folder:

In this example, the images shown left to right are named **ic_vol_none.png**, **ic_vol_full.png**, **bg_volumebar.png**, and **fill_volumebar.png**.

2. In the **Projects** view, right-click the `QtHmi` folder, then choose **Add Existing Files**.
3. In the file selector, select the files of the four images and click **Open**.

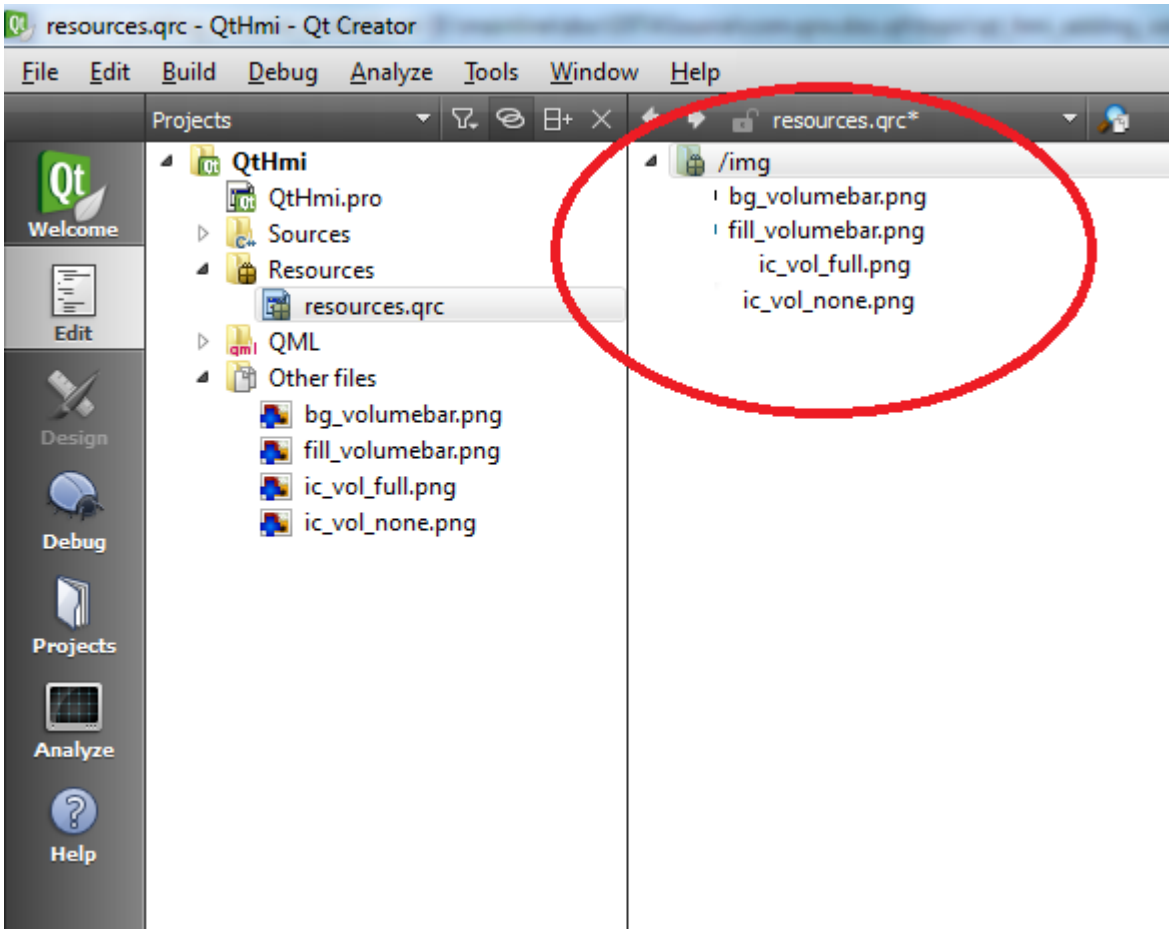
   A new folder, `Other files`, appears in the project view. This folder contains the four new image files.

4. Open the **resources.qrc** file for editing, by right-clicking its entry in the **Projects** view, then selecting **Open in Editor**.
5. In the configuration area near the bottom, click **Add**, then choose **Add Prefix**.



6. In the **Prefix** field, enter `img`.
7. Click **Add** again, then choose **Add Files**.
8. In the file selector dialog, select the files of the four images, then click **Open**.

   In the main editing area, the list of project resources now includes a prefix entry labelled `/img` and four file listings under the prefix.



The volume indicator and adjustment images are now part of your HMI project. Qt Creator will compile the images into the binary and your HMI can display them.

Adding the QML components

The **VolumeUI** and **VolumeSlider** components use QML to define the UI for the audio volume control. This UI consists of a slider indicating the current volume level and two buttons on the sides that increase and decrease the volume. You can tap the slider in a certain spot to set the volume to that exact level.

To add the QML components:

1. In the **Project** view, right-click the `QtHmi` folder and click **Add New...**
2. In the **New File** dialog, select `Qt` in the **Files and Classes** list, then `QML File (Qt Quick 2)` in the list of file types (shown in the middle), then click **Choose...**
3. In the **Location** page of the resulting dialog, name the file `VolumeUI`, then click **Next**.
4. In the **Summary** page, ensure the **Add to project** field is set to project file (**QtHmi.pro**), then click **Finish**.

   Qt Creator adds **VolumeUI.qml** to the project (under the **QML** folder) and opens this file for editing.

5. Replace the contents of this file with the following:

```
import QtQuick 2.0

Rectangle {
    id: root
    color: "#404040"
    width: parent.width
    height: parent.height / 8

    Row {
        id: volumeRow

        anchors.right: root.right
        anchors.rightMargin: root.width / 16
        anchors.verticalCenter: root.verticalCenter

        Item {
            id: volumeNone
            height: root.height
            width: height
            Image {
                id: volumeNoneImage
                anchors.centerIn: parent
                source: "qrc:/img/ic_vol_none.png"
            }
            Timer {
                id: volumeNoneTimer
                interval: 100
                repeat: true
                running: false
                onTriggered: {
                    // Decrease volume by 1%
                    volumeRow.updateVolumeSlider(
                        volumeSlider.value - 1)
                }
            }
            MouseArea {
                anchors.fill: parent
                onClicked: {
                    // Decrease volume by 1%
                    volumeRow.updateVolumeSlider(
                        volumeSlider.value - 1)
                }
                onPressAndHold: {
                    volumeNoneTimer.start();
                    // Decrease volume by 1%
```

```
                volumeRow.updateVolumeSlider(
                    volumeSlider.value - 1)
            }
            onReleased: {
                volumeNoneTimer.stop();
            }
        }
    }

    VolumeSlider {
        id: volumeSlider
        width: root.width / 4
        height: volumeNoneImage.height

        anchors.verticalCenter: parent.verticalCenter
        sourceBackground: "qrc:/img/bg_volumebar.png"
        sourceOverlay: "qrc:/img/fill_volumebar.png"
        value: 50
        maxValue: 100
    }

    Item {
        id: volumeFull
        height: root.height
        width: height
        Image {
            id: volumeFullImage
            anchors.centerIn: parent
            source: "qrc:/img/ic_vol_full.png"
        }
        Timer {
            id: volumeFullTimer
            interval: 100
            repeat: true
            running: false
            onTriggered: {
                // Increase volume by 1%
                volumeRow.updateVolumeSlider(
                    volumeSlider.value + 1)
            }
        }
        MouseArea {
            anchors.fill: parent
            onClicked: {
                // Increase volume by 1%
                volumeRow.updateVolumeSlider(
                    volumeSlider.value + 1)
```

```
                }
                onPressAndHold: {
                    volumeFullTimer.start()
                    // Increase volume by 1%
                    volumeRow.updateVolumeSlider(
                        volumeSlider.value + 1)
                }
                onReleased: {
                    volumeFullTimer.stop();
                }
            }
        }

        function updateVolumeSlider(value) {
            if (value > 100) {
                value = 100
            }
            if (value < 0) {
                value = 0
            }
            volumeSlider.value = value;
        }
    }
}
```

6. Repeat Steps 1 through 4 to add another QML file but this time, name the file `VolumeSlider`.
7. Replace the contents of this file with the following:

```
import QtQuick 2.0

// You need to specify the background image and the overlay
Item {
    id: root

    property string sourceBackground: ""
    property string sourceOverlay: ""

    // Max value
    property double maxValue: 0
    // Current value
    property double value: 0

    // Whether this item is user interactive
    property bool interactive: true

    Column {
```

```
spacing: 1
anchors.verticalCenter: parent.verticalCenter

Item {
    id: graphicBar

    width: root.width
    height: root.height;

    Image {
        id: sourceImage

        anchors.fill: graphicBar
        fillMode: Image.Tile
        smooth: true
        source: sourceBackground
    }

    Image {
        id: overlayImage

        height: graphicBar.height
        width: handle.x

        fillMode: Image.Tile
        smooth: true
        source: sourceOverlay
    }

    Item {
        // Invisible handle for dragging
        // The item doesn't need a width or height
        // because its x value is all that matters
        id: handle
        x: (maxValue ?
                (Math.min(value, maxValue) / maxValue)
                 * graphicBar.width : 0)
        width: 0
        height: 0
    }

    MouseArea {
        anchors.centerIn: parent

        height: parent.height * 3
        width: parent.width
```

```
                    enabled: root.interactive

                    drag.target: handle
                    drag.minimumX: 0
                    drag.maximumX: graphicBar.width

                    function moveToPosition(position)
                    {
                        if (!maxValue)
                            return;
                        // retrieve the position where the user
                        // dragged to
                        value = (position / graphicBar.width)
                                 * maxValue
                    }

                    // Touch without drag
                    onReleased: {
                        moveToPosition(mouseX);
                    }

                    property bool dragActive: drag.active

                    onPositionChanged: {
                        moveToPosition(handle.x);
                    }
                }
            }
        }
    }
}
```

8. Open **main.qml** and update its contents with the following:

```
import QtQuick 2.0

Rectangle {

    color: "black"

    Text {
        color: "white"
        text: qsTr("Awesome HMI goes here")
        anchors.centerIn: parent
    }

    VolumeUI {
        id: volumeui
```

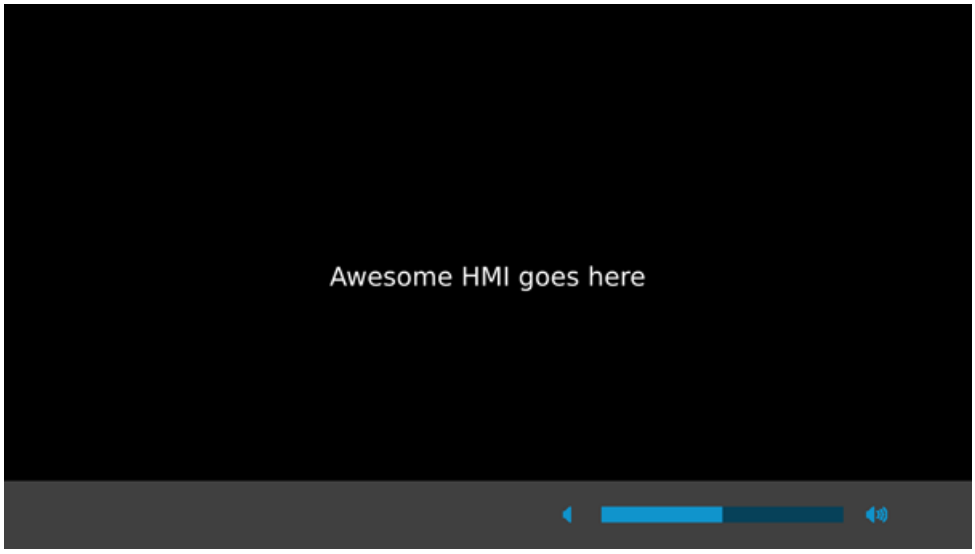
```
            anchors.left: parent.left
            anchors.right: parent.right
            anchors.bottom: parent.bottom
        }
}
```

This adds the volume control to the bottom of the HMI.

9. Build and run the HMI application, by following the steps in "*Building the HMI application for a QNX target* (p. 78)".

   The HMI shown on the target screen prints the original message but also displays the volume slider and two control buttons along the bottom. Clicking the left button decreases the volume by 1% and moves the slider to the left. Clicking the right button increases the volume by 1% and moves the slider to the right. Tapping the slider sets the volume to the exact level based on the location. For instance, tapping it in the middle sets the volume to 50%.

   You can also drag the volume slider to the left to decrease the volume or to the right to increase it. Whenever your tap or drag the slider, the volume level is redrawn immediately and the audio status PPS object is updated to store this new level.



You've now added an HMI control for setting the audio volume!

The control defined here lets the user interact with the volume display and keeps the PPS volume setting in sync with the HMI, but it doesn't tell the Audio Manager service to change the output volume. To do this, your application has to write a command to the PPS control object used by the Audio Manager service (for more information, see the **/pps/services/audio/status** entry in the *PPS Objects Reference*).

Index

S

T

W