Application Note

Building Modern GUIs with Qt 6 on TI Embedded Platforms

Shriya Surti, Krunal Bhargav

ABSTRACT

Qt is a powerful, cross-platform framework for developing graphical user interface (GUI) applications. This application note presents a step-by-step example that demonstrates how to create a simple Qt-based GUI, use the GUI to blink an LED, cross-compile the project for the AArch64 architecture, and deploy the GUI on a Texas Instruments (TI) platform. The document covers the complete workflow, including setting up the Qt environment, developing a basic application, building the application for the target architecture, and running the application on the AM62P platform.

Table of Contents

1 Introduction......2
2 Detailed Description......2
2.1 Launching Qt Creator......2
2.2 Running The First Project on PC......6
2.3 Creating a Button......8
2.4 Creating a C++ Back-End Component......11
2.5 Connecting to the EVM......16
2.6 Building and Deploying Your Application For Your Embedded Platform......16
2.7 Running the Application......17
3 Summary......19
4 References......19

Trademarks

All trademarks are the property of their respective owners.

1 Introduction

This walkthrough uses a TI EVM: SK-AM62P-LP evaluation board

Note

This tutorial also works for any AM62x/AM62P.

2 Detailed Description

Prerequisites

Start by making sure Ubuntu 22.04 is being used.

Install the QEMU user mode emulators for enabling cross-architecture execution on host system:

sudo apt-get install qemu-user-static

qtcreator is required to create the Qt application.

sudo apt install qtcreator

Use the following example to understand how to Blink an LED on the AM62P/AM62X: Blinking an LED

2.1 Launching Qt Creator

To begin creating the application, launch Qt Creator by running:

qtcreator

Figure 2-1 shows the screen after launching Qt Creator.

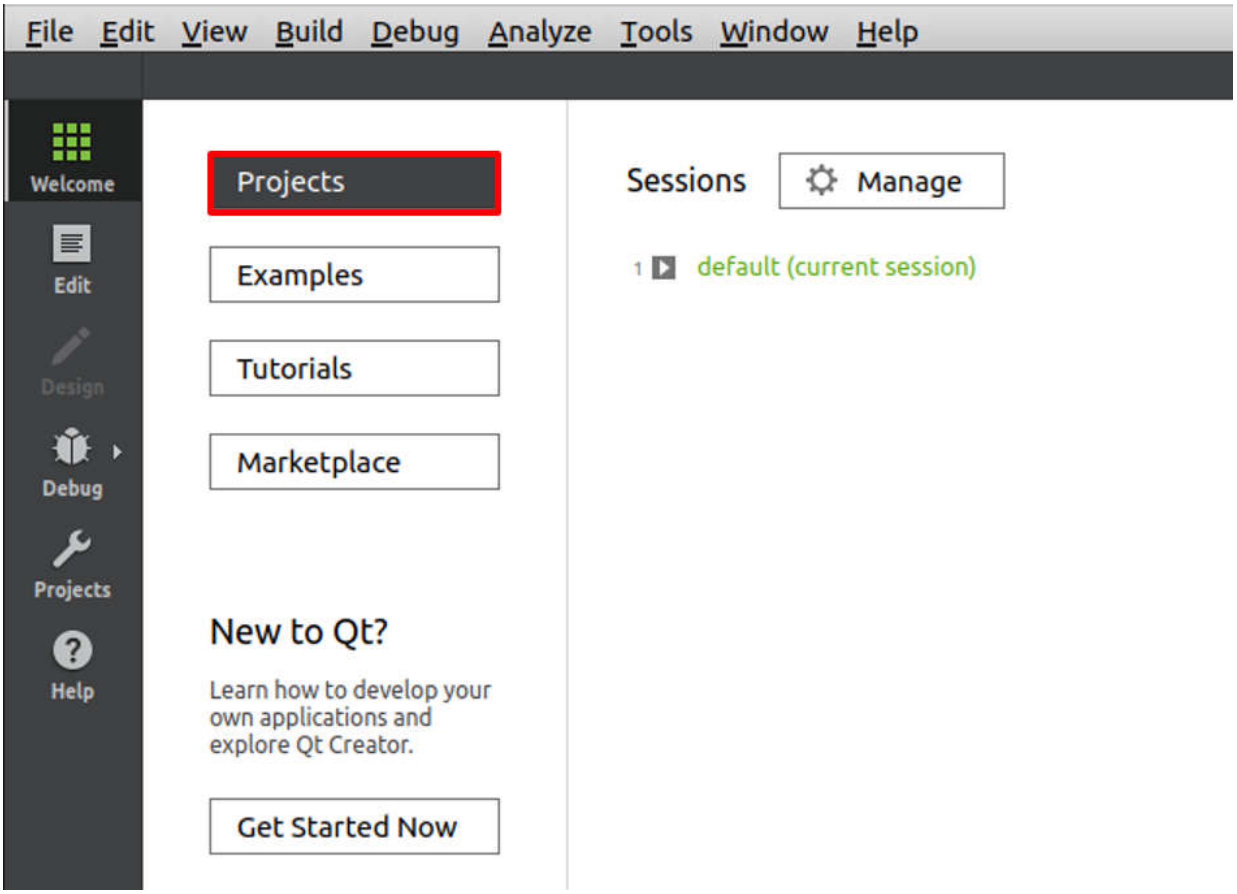
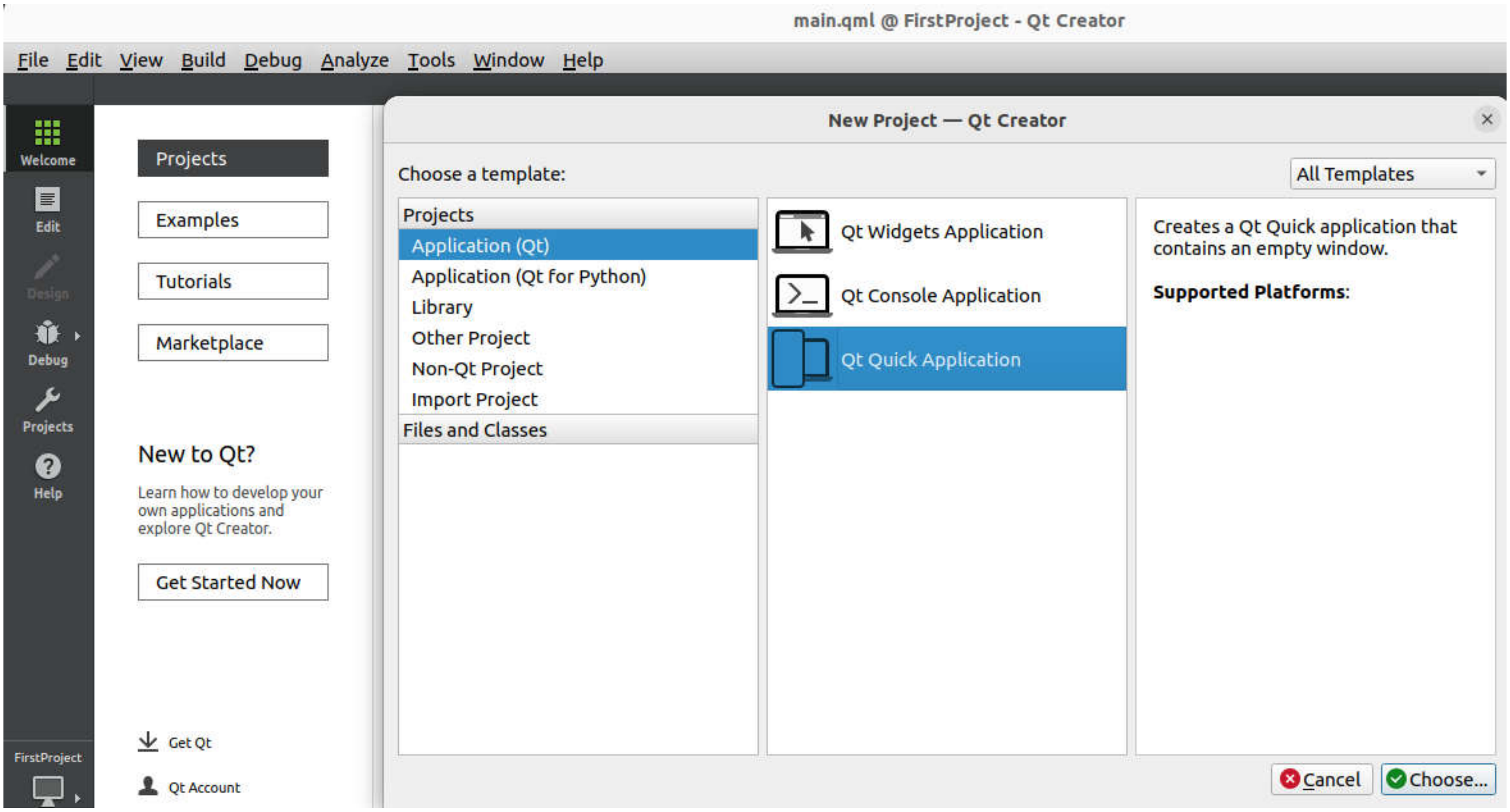


Figure 2-1. Starting a New Project

From there, create a new project by clicking the *Project* button shown in Figure 2-1.



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

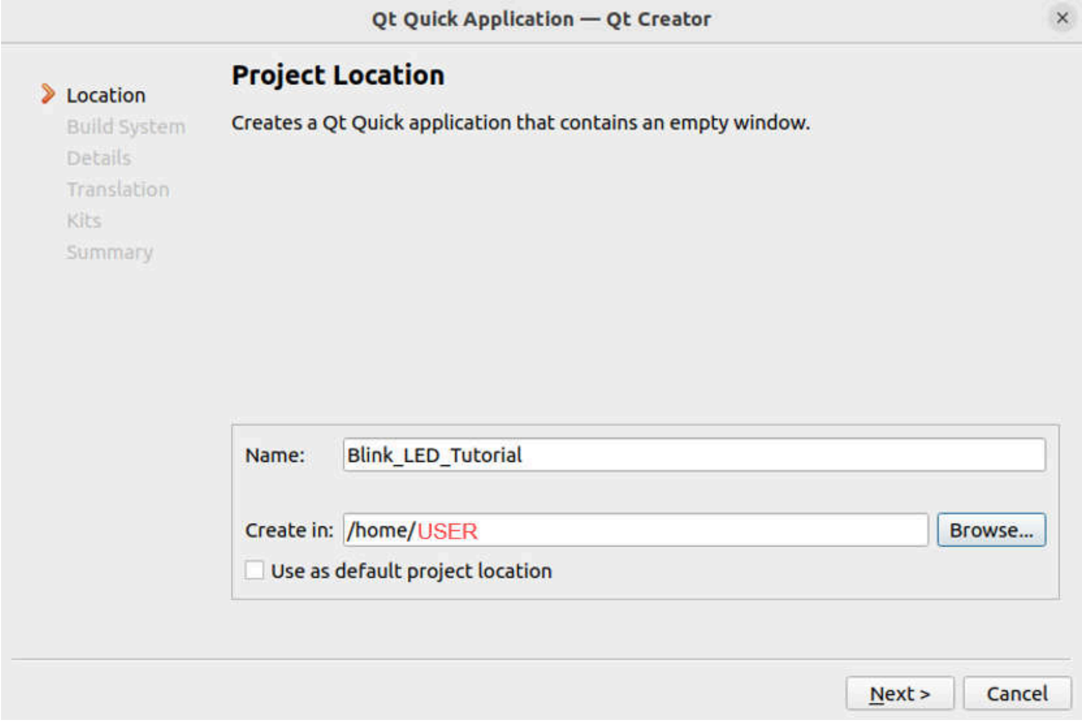
Creates a Qt Quick application that contains an empty window.

Supported Platforms:

Figure 2-2. Creating a Qt Application

Select Qt Quick Application. After clicking *Choose*, click *Next* until the following screens are showing. Select *cmake* and make sure to select Qt 6.X. In this case, the minimum was 6.2.

Next, specify a location to be under /home/USER (where user is the username) or to a selected location of the user.



Project Location

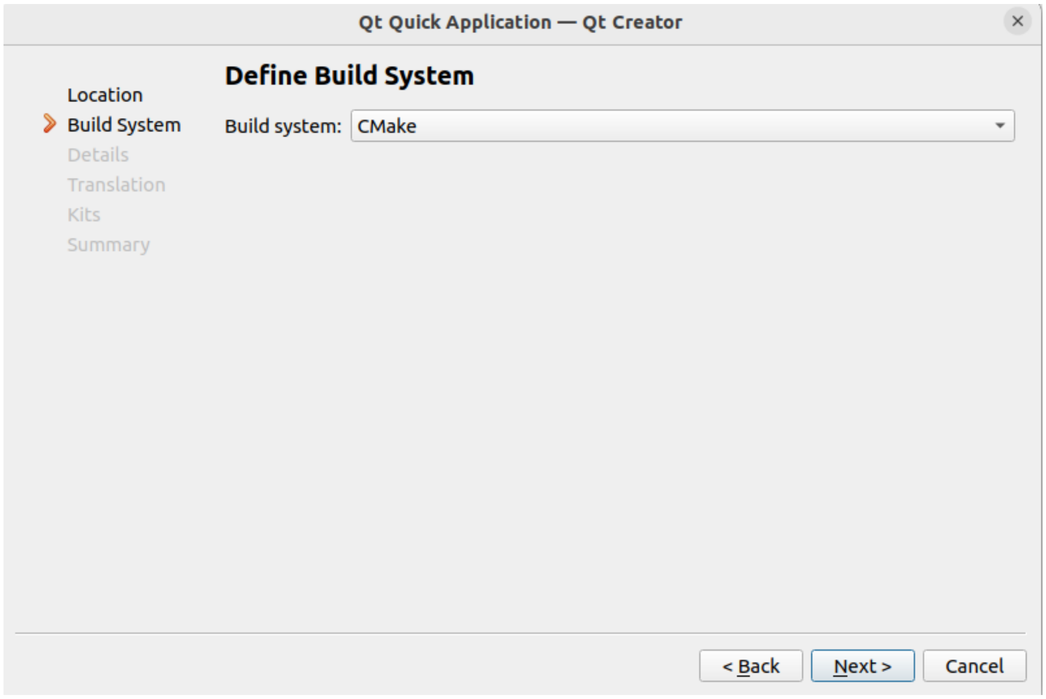
Creates a Qt Quick application that contains an empty window.

Name: Blink_LED_Tutorial

Create in: /home/USER

☐ Use as default project location

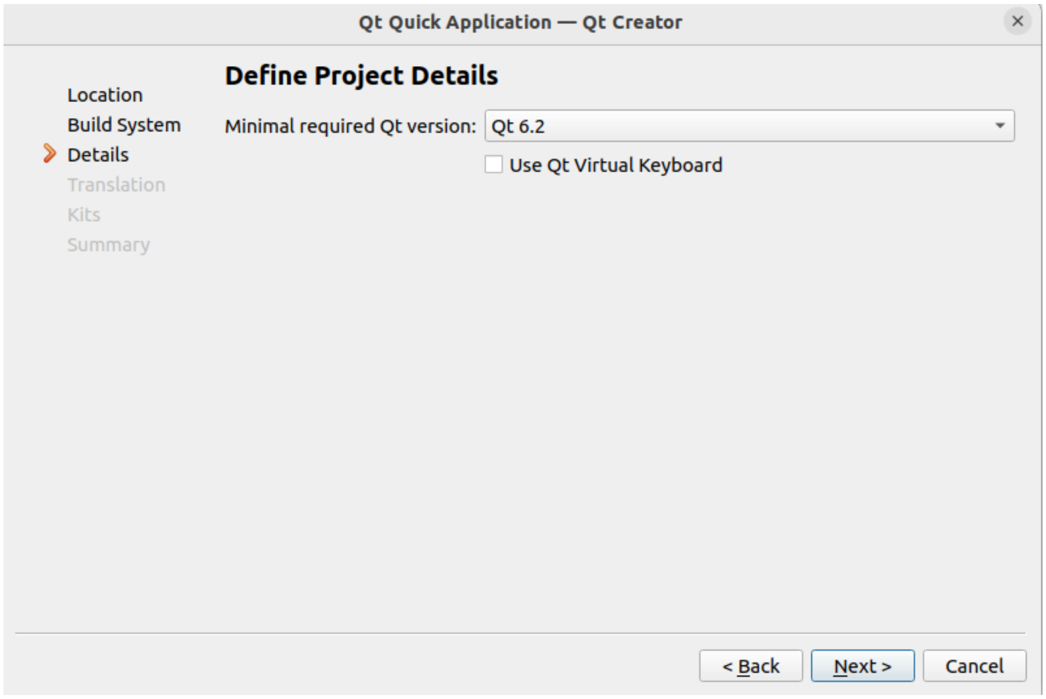
Figure 2-3. Specifying Project Location



Define Build System

Build system: CMake

Figure 2-4. Using CMake



Define Project Details

Minimal required Qt version: Qt 6.2

□ Use Qt Virtual Keyboard

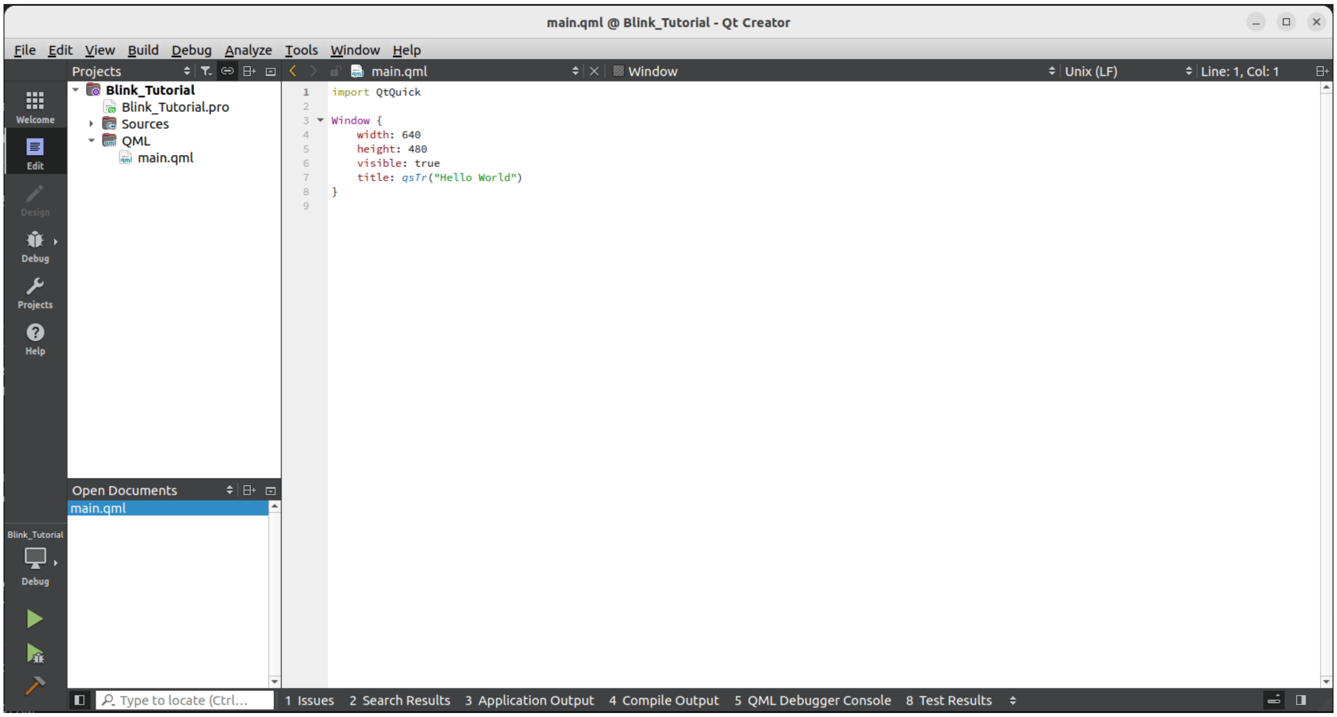
Figure 2-5. Specifying Qt Version

Then continue through the settings until clicking *Finish*.

Once the project is created, the user has a basic Qt application.

2.2 Running The First Project on PC

The workplace must look similar to the example shown in Figure 2-6

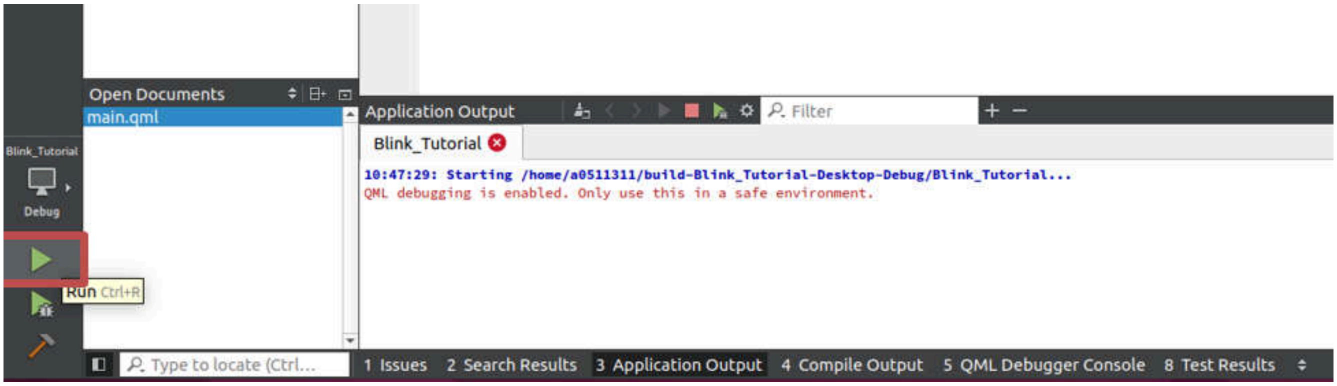

main.qml @ Blink_Tutorial - Qt Creator
File Edit View Build Debug Analyze Tools Window Help
Projects
Blink_Tutorial
Blink_Tutorial.pro
Sources
QML
main.qml
Welcome
Edit
Design
Debug
Projects
Help
Open Documents
main.qml
Blink_Tutorial
Debug
Window
Unix (LF)
Line: 1, Col: 1
Type to locate (Ctrl...
1 Issues 2 Search Results 3 Application Output 4 Compile Output 5 QML Debugger Console 8 Test Results

```
import QtQuick

Window {
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")
}
```

Figure 2-6. Baseline of Project

Figure 2-6 creates a window. This tutorial does not discuss Window declarations. To learn more about the window declaration, see the following: Window QML Type | Qt Quick | Qt 6.10.1

Continue and click the *Run* button.



```
10:47:29: Starting /home/a0511311/build-Blink_Tutorial-Desktop-Debug/Blink_Tutorial...
QML debugging is enabled. Only use this in a safe environment.
```

Figure 2-7. Running Your Application

When the application runs, a screen displays with the title *Hello World* because a title has been defined for the window.

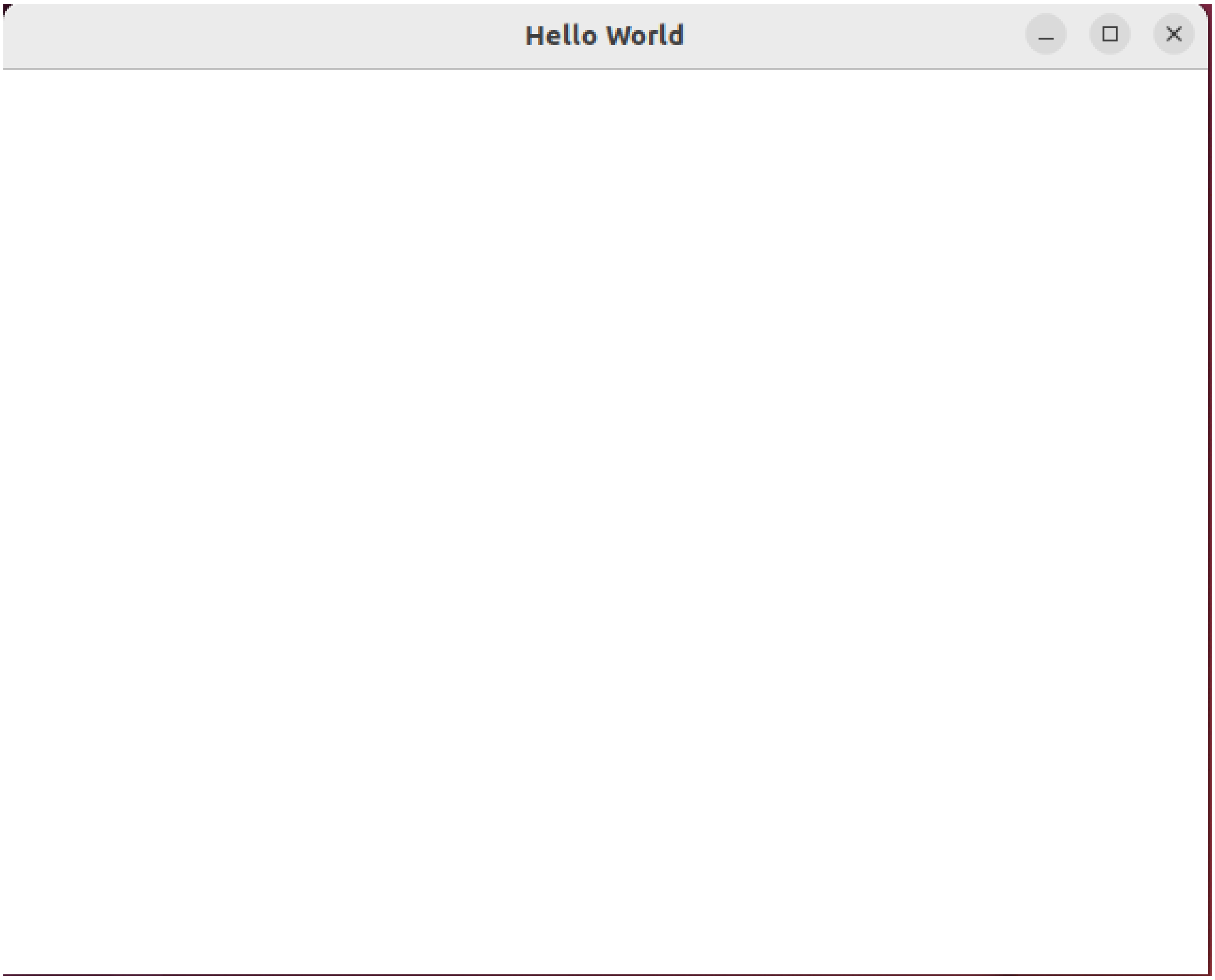


Figure 2-8. Hello World Window

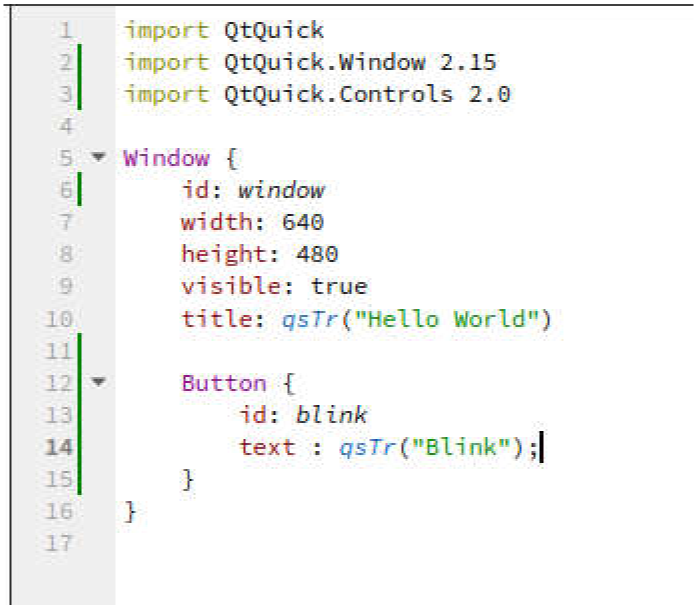
The window size appears as 480×640, which matches the dimensions specified in the Window component.

The next step is to add a button named *Blink* to the application.

2.3 Creating a Button

In Qt QML, a button is created using the *Button* component, which is part of the *QtQuick.Controls* module. A component in QML is simply a reusable building block, such as a window, label, button, image, or a custom UI element that is defined.

Components encapsulate both behavior and appearance, making the components simple to use and organize in a UI layout. For custom components, see the following list by QT: The QML Reference | Qt Qml | Qt 6.10.2



```
import QtQuick
import QtQuick.Window 2.15
import QtQuick.Controls 2.0

Window {
    id: window
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")

    Button {
        id: blink
        text : qsTr("Blink");
    }
}
```

Figure 2-9. Adding a Button

The id gives the button a unique reference so the program can refer back to the button. For example, handling signals, checking the state, or modifying the properties.

The text property sets the label that appears on the button.

When a user clicks *Run*, the following is visible:

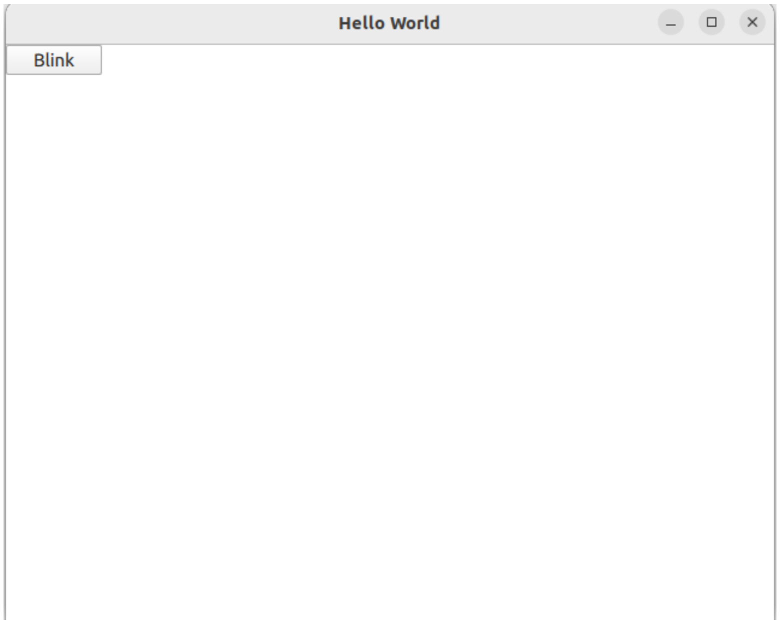
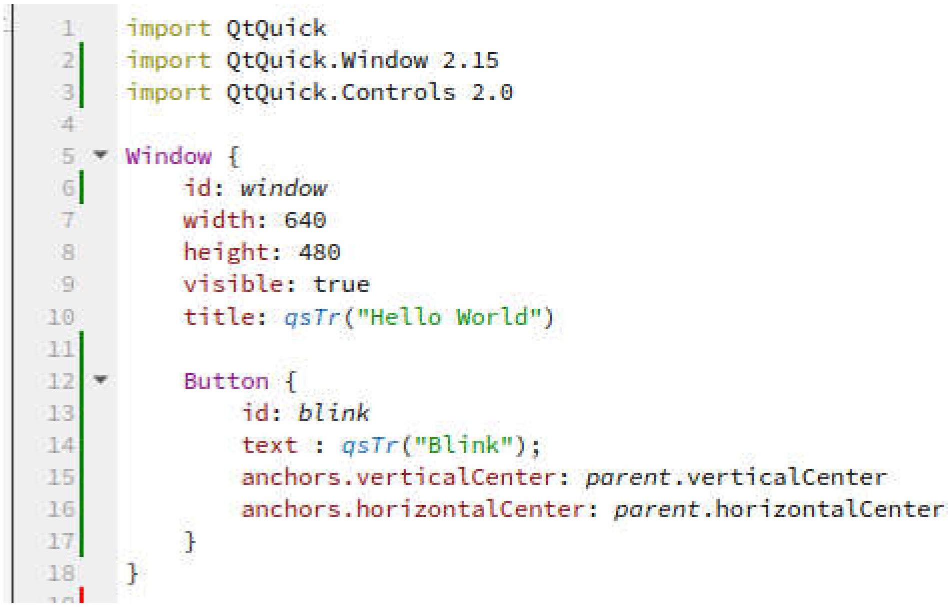


Figure 2-10. Modified Window with Blink Button

There is now a button within a window called *Blink*.

2.3.1 Modifying the Property of a Button

Moving the button in the center of the screen is possible.



```
import QtQuick
import QtQuick.Window 2.15
import QtQuick.Controls 2.0

Window {
    id: window
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")

    Button {
        id: blink
        text : qsTr("Blink");
        anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter
    }
}
```

Figure 2-11. Repositioning Blink Button

Use anchors. Anchors is a property within QML that allows UI elements to be positioned relative to other elements. A parent is something that can be used without manually calculating coordinates pixel by pixel. It helps define relationships between items such as the window and button. In this case, the *Blink* button is centered in the window.

Now when a user runs the application, the button is now centered on the screen.

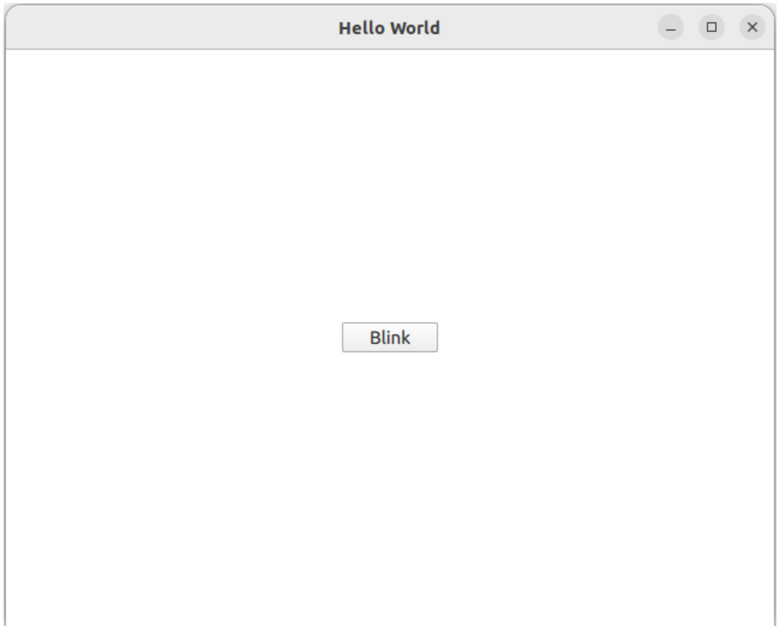
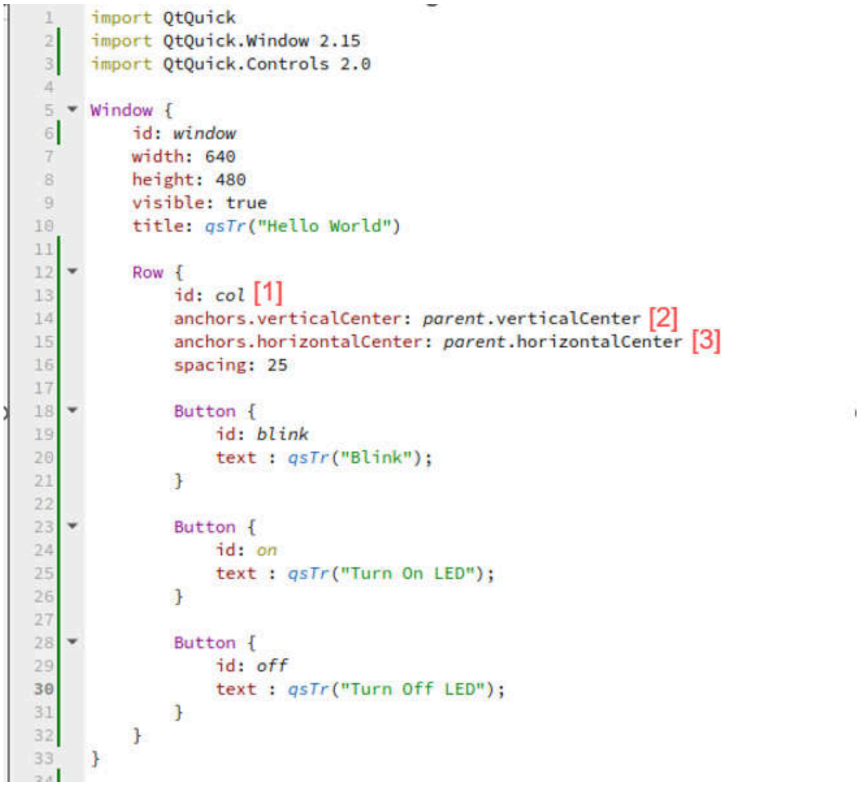


Figure 2-12. Button Repositioned in Window

Can there be more than one button in the same row?



```
import QtQuick
import QtQuick.Window 2.15
import QtQuick.Controls 2.0

Window {
    id: window
    width: 640
    height: 480
    visible: true
    title: qsTr("Hello World")

    Row {
        id: col
        anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter
        spacing: 25

        Button {
            id: blink
            text : qsTr("Blink");
        }

        Button {
            id: on
            text : qsTr("Turn On LED");
        }

        Button {
            id: off
            text : qsTr("Turn Off LED");
        }
    }
}
```

Figure 2-13. Adding a Row Component

Modify the program by adding a component called Row [1]. Row is a tool to help organize all the components in one row. In this case, components are organized along the center row utilizing the vertical [2] and horizontal [3] center of the window. Now any components that are added within this scope are on the same row and centered as shown Figure 2-14.

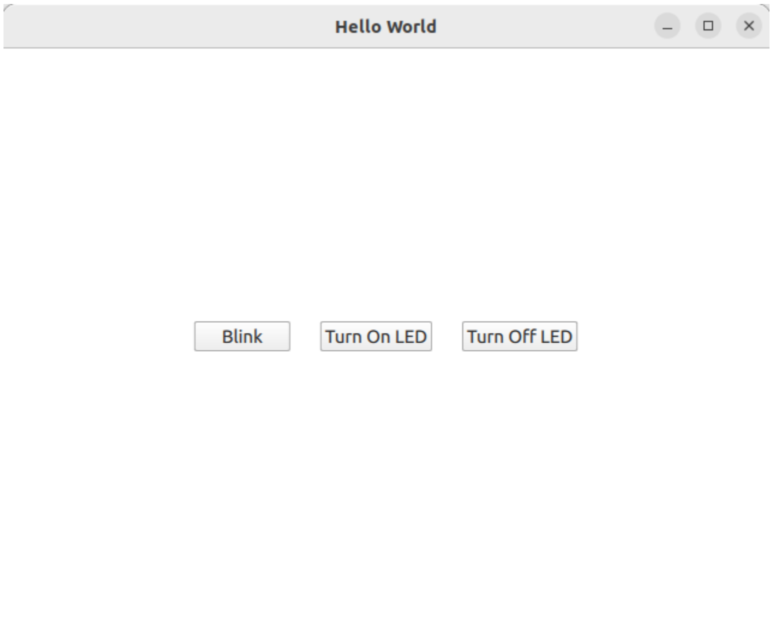


Figure 2-14. Multiple Components Centered Within Window

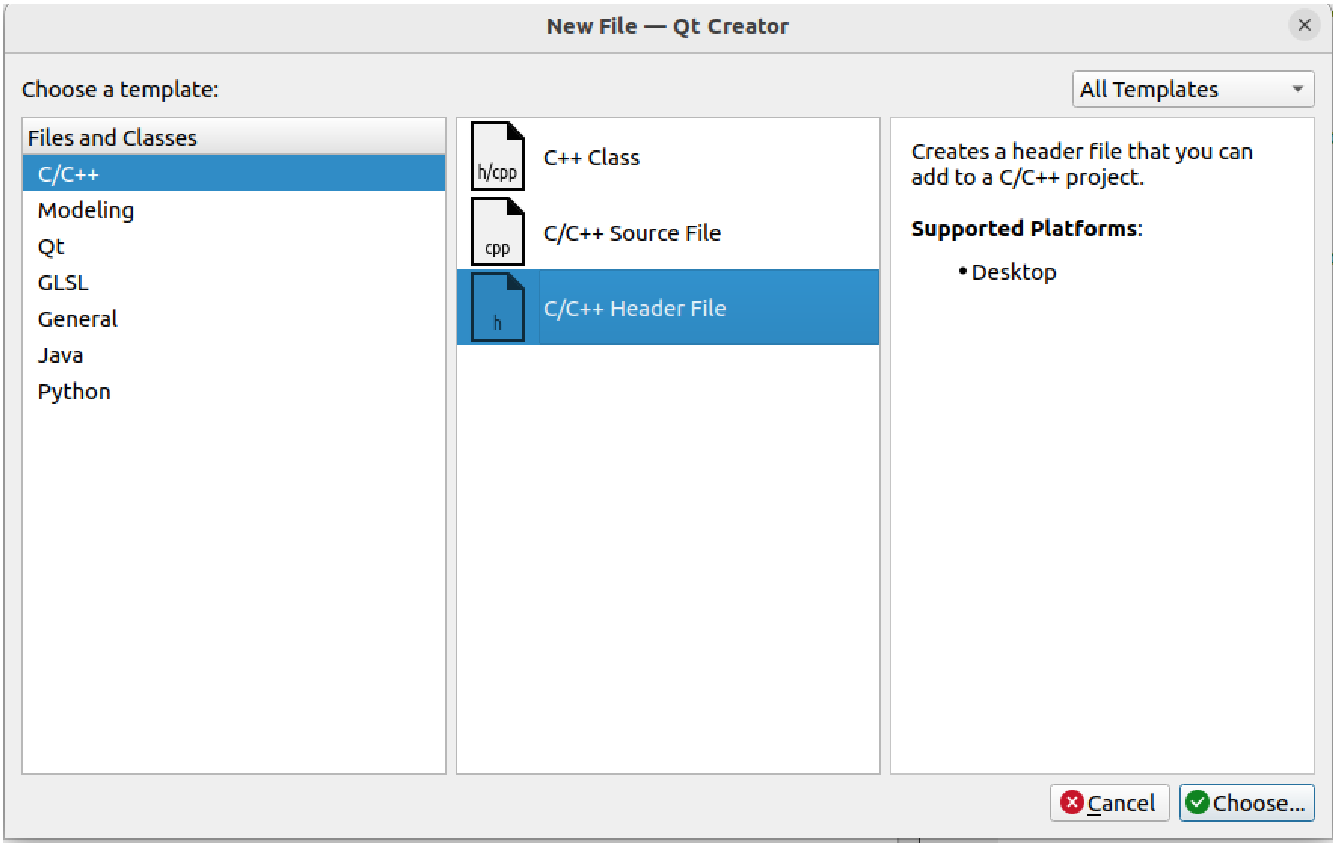
2.4 Creating a C++ Back-End Component

Now that the button is created in the QML file, add functionality to them so that the button can produce some results when clicked. To achieve this, write a C++ application that handles the back-end components of the project.

This is a common approach in Qt development, where a user uses QML for the front-end (user interface) and C++ for the back-end (business logic). By separating the front-end and back-end, a user keeps code organized and maintainable.

2.4.1 Defining the Scope of the Project

First, create a new file within the scope of the project.



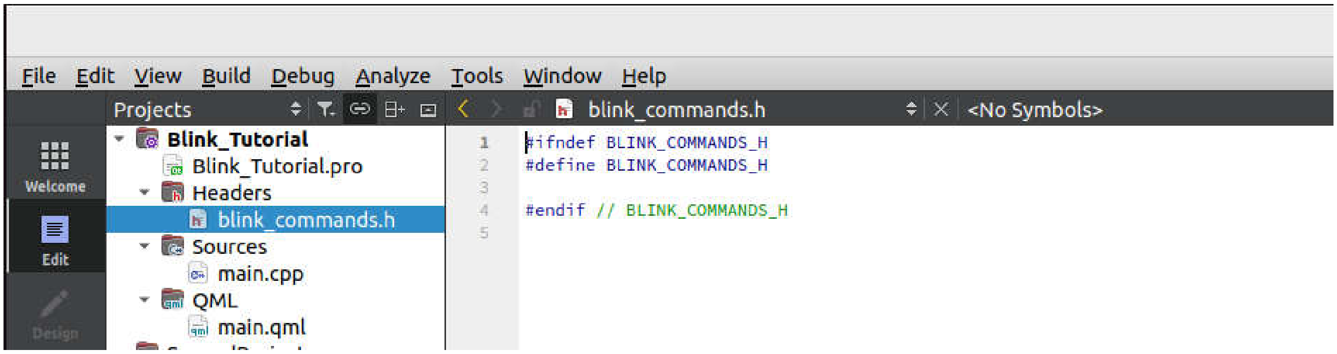
Creates a header file that you can add to a C/C++ project.

Supported Platforms:

- Desktop

Figure 2-15. Creating a Header File

With the file created, the screen looks like the following:

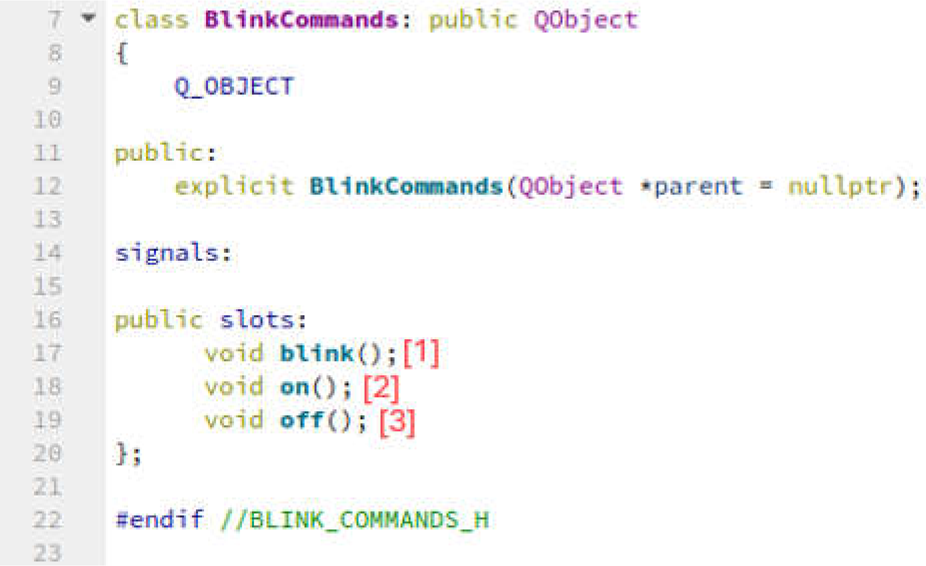


```
#ifndef BLINK_COMMANDS_H
#define BLINK_COMMANDS_H

#endif // BLINK_COMMANDS_H

```

Figure 2-16. Baseline Header File

In Qt, a *slot* is a function that runs when something happens in the program. This works together with a *signal*, which is sent by an object to let the program know that an event has occurred, such as a button being clicked.

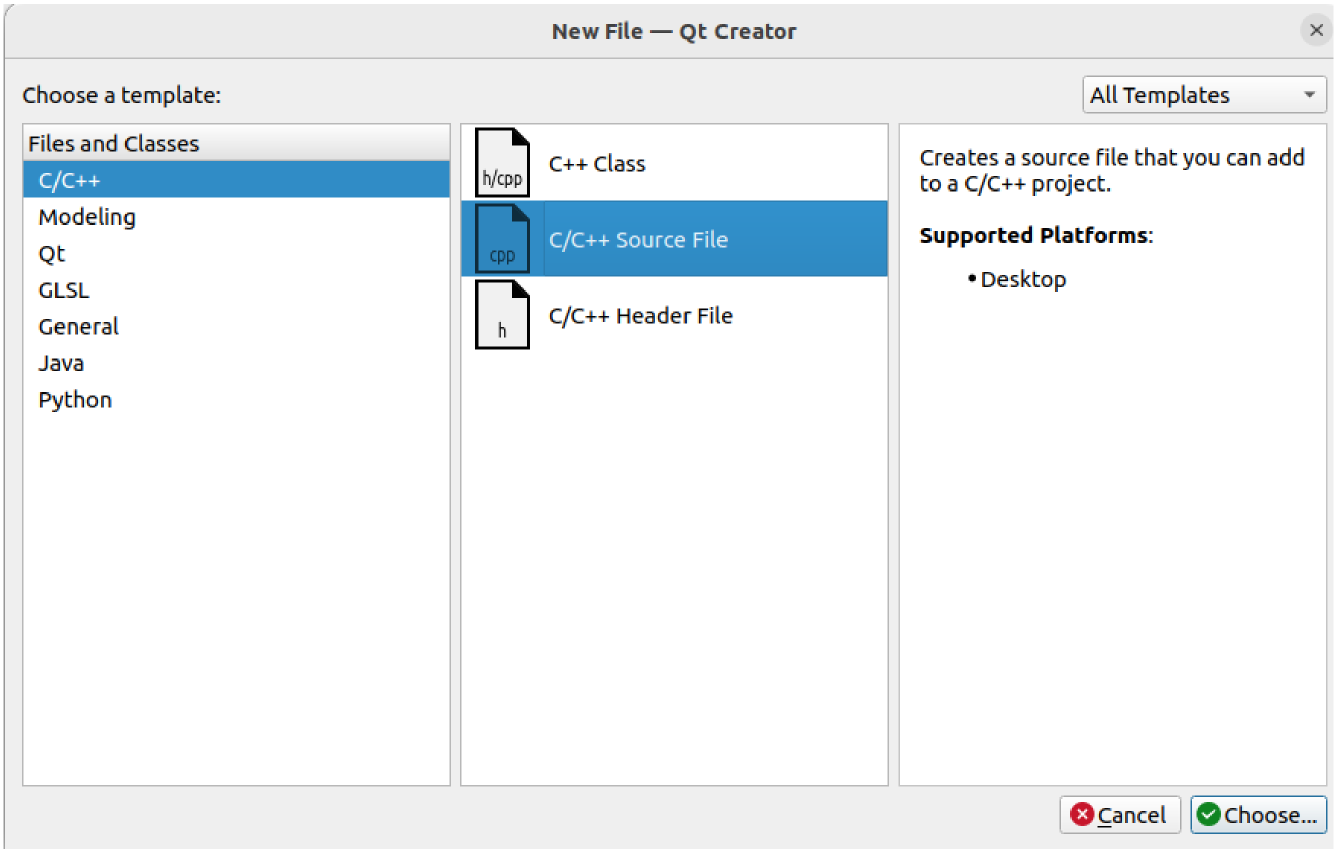
Think of the signal as a message, and a slot as the function that receives and responds to that message. In this header file, what happens when the three buttons, *blink[1]* , *on[2]*, and *off[3]*, are clicked is defined. Each button is connected to a slot, and when the user clicks a button, the corresponding slot is called to perform the required action.



```
class BlinkCommands: public QObject
{
    Q_OBJECT

public:
    explicit BlinkCommands(QObject *parent = nullptr);

signals:

public slots:
      void blink();[1]
      void on(); [2]
      void off(); [3]
};

#endif //BLINK_COMMANDS_H

```

Figure 2-17. Creating Slots

Now that the header file has been created and the slots have been defined, create a C++ file to implement the slots. This C++ file contains the actual code that is executed when the slots are triggered.



Creates a source file that you can add to a C/C++ project.

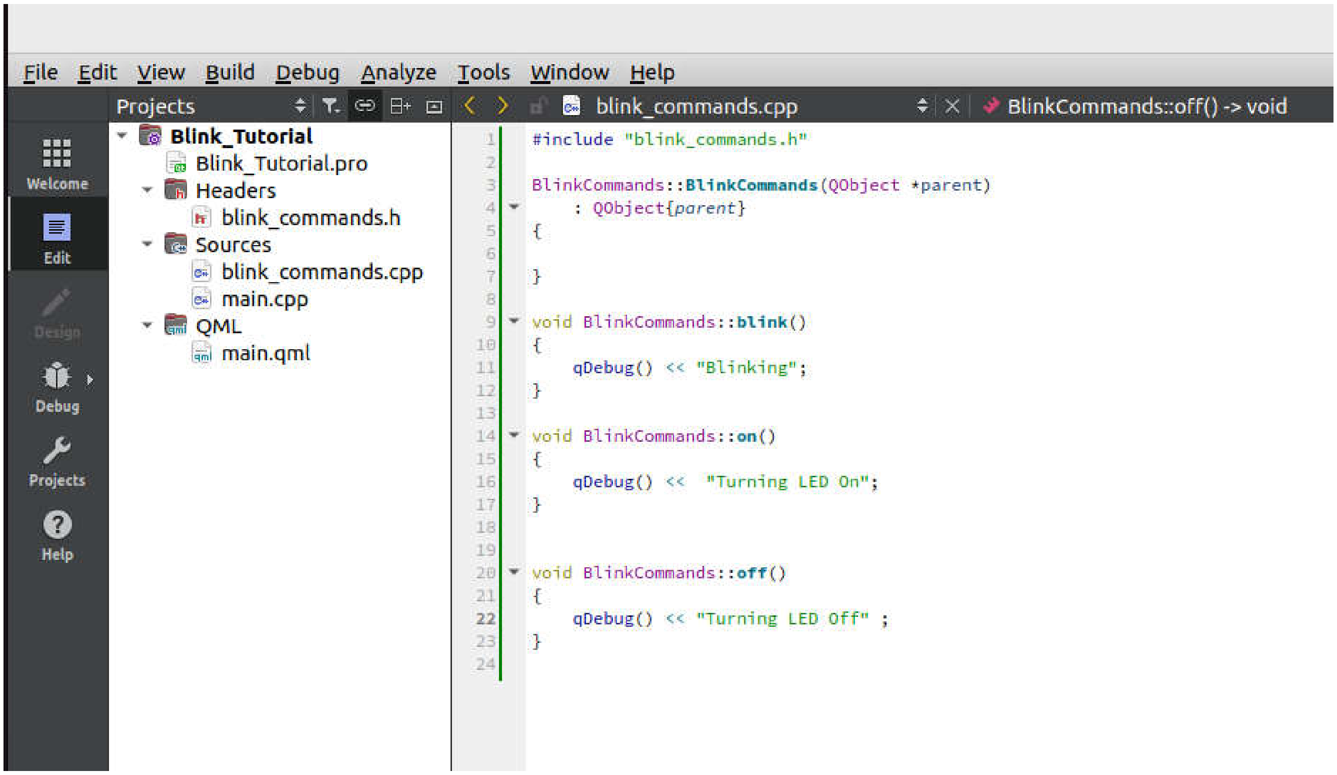
Supported Platforms:

- Desktop

Figure 2-18. Creating a Source File

Use the same naming convention used for the header file.

Utilizing QDebug, make simple print statements whenever each button is clicked:



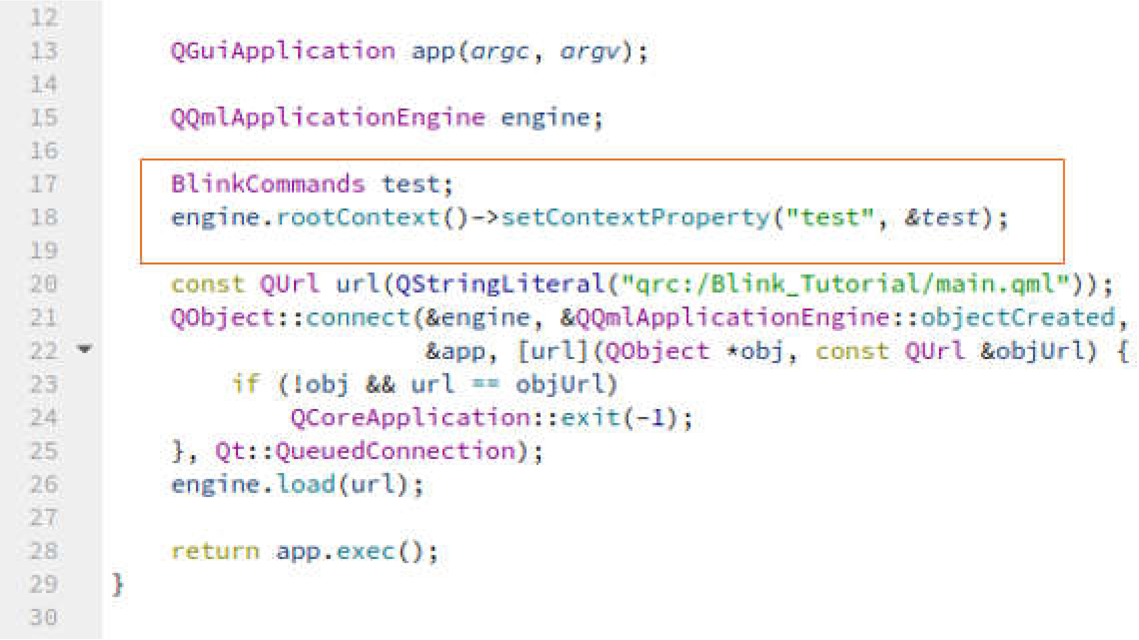
```cpp
#include "blink_commands.h"

BlinkCommands::BlinkCommands(QObject *parent)
    : QObject{parent}
{

}

void BlinkCommands::blink()
{
    qDebug() << "Blinking";
}

void BlinkCommands::on()
{
    qDebug() <<  "Turning LED On";
}


void BlinkCommands::off()
{
    qDebug() << "Turning LED Off" ;
}

```

Figure 2-19. Creating Outputs for Slots

These print statements are executed whenever each button is clicked and prints the corresponding message to the console.

2.4.2 Connecting Your C++ Application to Your QML Application

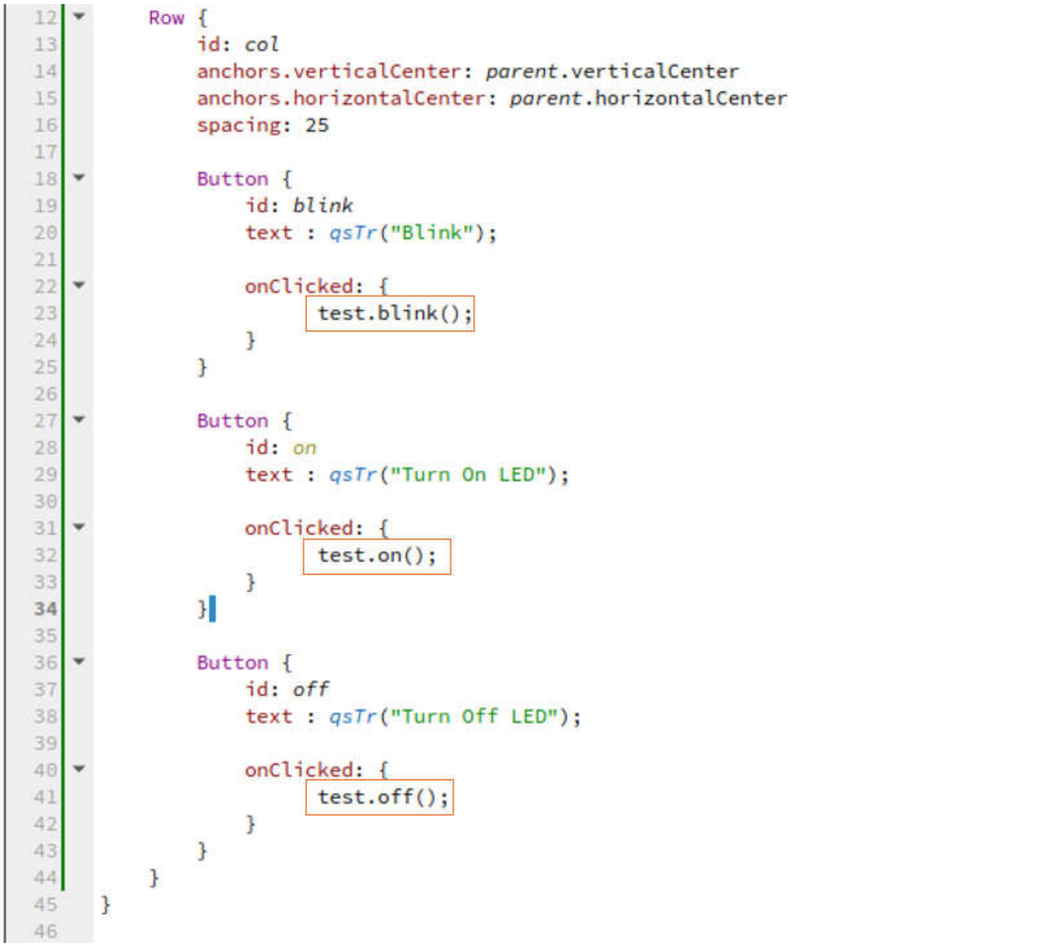
To connect the slots to the front-end, create an object in the main.cpp file. This object is going to be used in the qml file to call the functions defined in the back-end that was written.



```cpp

    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;

    BlinkCommands test;
    engine.rootContext()->setContextProperty("test", &test);

    const QUrl url(QStringLiteral("qrc:/Blink_Tutorial/main.qml"));
    QObject::connect(&engine, &QQmlApplicationEngine::objectCreated,
                     &app, [url](QObject *obj, const QUrl &objUrl) {
        if (!obj && url == objUrl)
            QCoreApplication::exit(-1);
    }, Qt::QueuedConnection);
    engine.load(url);

    return app.exec();
}

```

Figure 2-20. Connecting Slots to Your Object

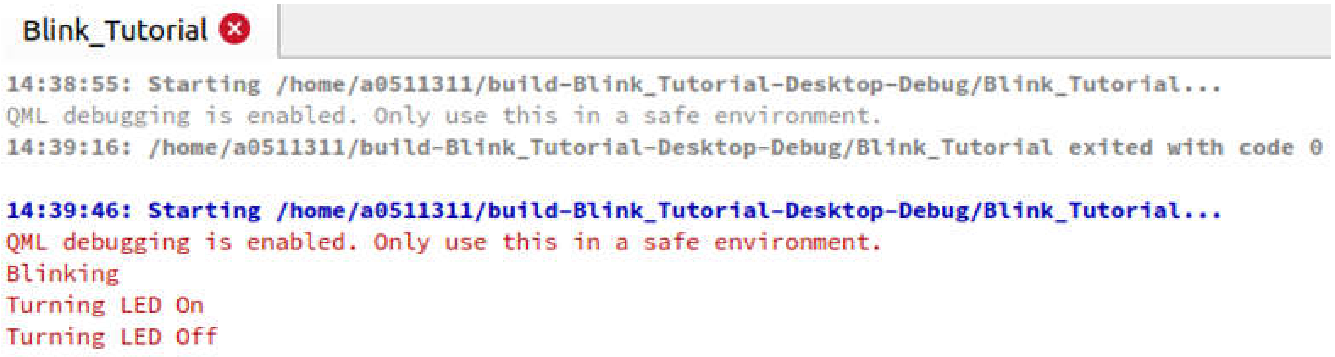
Use the onClicked() function within the QML file to make the QT application aware that the application must do something when a button is clicked. In this case, call the C++ functions, blink(), on(), and off(), that were defined in the back-end.



```
    Row {
        id: col
        anchors.verticalCenter: parent.verticalCenter
        anchors.horizontalCenter: parent.horizontalCenter
        spacing: 25

        Button {
            id: blink
            text : qsTr("Blink");

            onClicked: {
                test.blink();
            }
        }

        Button {
            id: on
            text : qsTr("Turn On LED");

            onClicked: {
                test.on();
            }
        }

        Button {
            id: off
            text : qsTr("Turn Off LED");

            onClicked: {
                test.off();
            }
        }
    }
}
```

Figure 2-21. Calling C++ Functions

Now when the user runs the application and clicks each of the three buttons in theWindow, the following results are shown in the terminal:



```
14:38:55: Starting /home/a0511311/build-Blink_Tutorial-Desktop-Debug/Blink_Tutorial...
QML debugging is enabled. Only use this in a safe environment.
14:39:16: /home/a0511311/build-Blink_Tutorial-Desktop-Debug/Blink_Tutorial exited with code 0

14:39:46: Starting /home/a0511311/build-Blink_Tutorial-Desktop-Debug/Blink_Tutorial...
QML debugging is enabled. Only use this in a safe environment.
Blinking
Turning LED On
Turning LED Off
```

Figure 2-22. Output of Each Function

2.5 Connecting to the EVM

Now, follow these steps to make the LEDs actually blink. Within the C++ application, go back and change the print statements within the blink_command.cpp file (or whatever the file is named) to use the LED on the board. For an AM62P, use the following commands:

Blinking an LED:

```
system("echo heartbeat > /sys/class/leds/am62-sk\\:green\\:heartbeat/trigger");
```

Turning an LED On:

```
system("echo 1 > /sys/class/leds/am62-sk\\:green\\:heartbeat/brightness");
```

Turning an LED Off:

```
system("echo none > /sys/class/leds/am62-sk\\:green\\:heartbeat/trigger");
```

Note: If the user is using a different board, the previous statement can differ slightly.

Do not be concerned about the terminal output for now, since the program is not running on the board. The program does not understand the previous commands until the program is on the board.

2.6 Building and Deploying Your Application For Your Embedded Platform

The first Qt application is now completed. The next step is to build the project and produce an executable that can run on the AM62P (or other AM6x) board.

2.6.1 Build the Application

To make the build process easier, use a docker image. Here are the steps to build through a docker image. First, cd into the project directory. Use Blink_Tutorial as an example.

Pull TI's debian-arm64 Docker image and run the image:

```
docker pull ghcr.io/texasinstruments/debian-arm64:latest
```

```
docker run -it -v ${PWD}:/root/workspace ghcr.io/texasinstruments/debian-arm64:latest bash
```

Inside the container, make the project with the following:

```
cd /root/workspace
```

```
cmake -B build -S .
```

```
make -C build
```

This creates an executable within the build folder that a user can then send to the AM62P.

2.6.2 Transfer Executable to the Board

Copy the executable over Ethernet using:

```
scp file_name root@ip_address:/destination/path
```

2.7 Running the Application

Before launching the Qt application, enable weston. To do so, export the Wayland display:

export WAYLAND_DISPLAY=/run/user/1000/wayland-1

Now run the applcation: *./ProjectName*

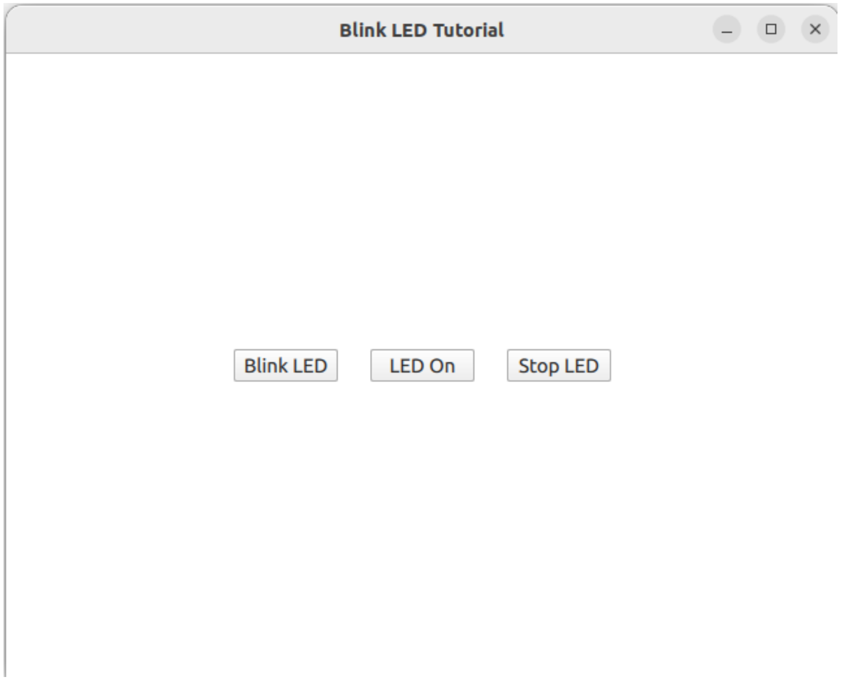


Figure 2-23. Running Your Application on Your Board

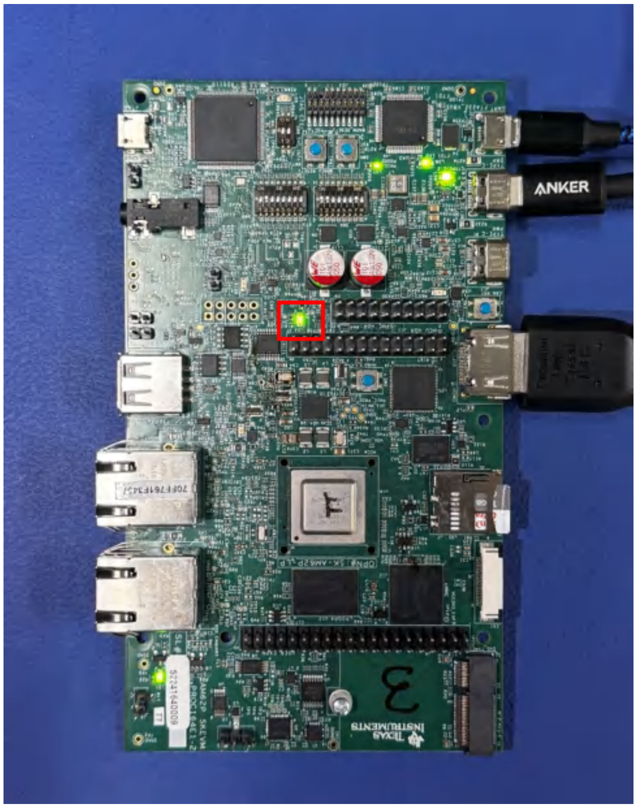


Figure 2-24. Blinking LED On Your Board

Congratulations, the Qt project must now start and display on the board!

When clicking each of the buttons, the green LED shown above either blinks, stays on, or turns off.

3 Summary

After completing these steps, the user must be able to develop a GUI through Qt and deploy the GUI on an embedded platform. Now that the basics have been explained, a user can now develop a more complex GUI.

4 References

1. Texas Instruments, *Blinking an LED* , resource explorer.
2. Texas Instruments, *1. Overview — Processor SDK AM62Px Documentation*, webpage.
3. Texas Instruments, *Qt Framework – Build Fast, Scalable Cross-Platform Software | Qt*, webpage.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025