Qt for MCUs application using the Qt Creator and Renesas e^2 studio IDEs Introduction

Qt for MCUs offers all the necessary tools to design, develop, build, and deploy your application onto the target.

This guide helps you understand the complete cycle of developing a Qt for MCUs application using Qt Design Studio, Qt Creator, and Renesas e^2 studio. It is organized into several chapters that describe the process and provide step-by-step instructions for the following tasks:

- Design a simple UI using Qt Design Studio
- Create an interface object and build the application as a static library using Qt Creator.
- Create an e^2 studio project and import the static library.
- Configure the FSP for the target and make necessary changes to the C++ code.
- Build and debug the project on the target.

The instructions in this guide are relevant for the Renesas EK-RA6M3G reference platform. Before you begin, install the software prerequisites listed below and set up your development host.

Prerequisites

The following software prerequisites must be installed to start developing for the Renesas EK-RA6M3G target platform on the Linux or Windows host. You can install most of these prerequisites using the Qt online installer, which is available for download from your Qt account.

Note: Qt for MCUs requires a license. If you do not have one, you can request a free trial at https://www.qt.io/contact-us/request-a-free-trial.

The following prerequisites are required irrespective of the development host you are on. Other versions may work but have not been tested.

- Qt for MCUs SDK v2.2
- Qt Design Studio v3.5.0
- Qt Creator v6.0.0 or newer
- CMake 3.15 or newer
- Ninja 1.10.0 or newer

In addition, the following are also required as they are not offered by the Qt online installer:

- Renesas e^2 studio IDE version 2022-04 or newer
- Renesas Flexible Software Package (FSP) v3.7.0
- SEGGER J-Link Software and Documentation pack v6.96 or newer

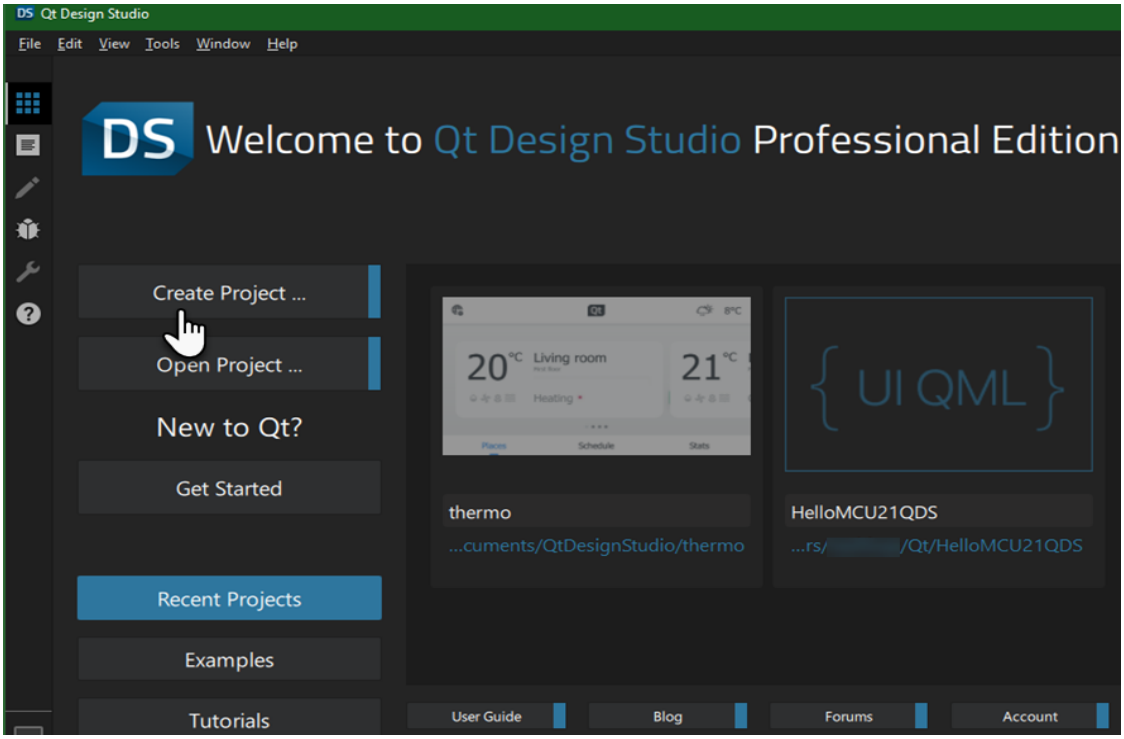
1. Design UI

This chapter provides step-by-step instructions to design a simple UI for the LED indicator application. You need Qt Design Studio to get started. If you don't have it installed, install it using the Qt Online Installer or Maintenance Tool.

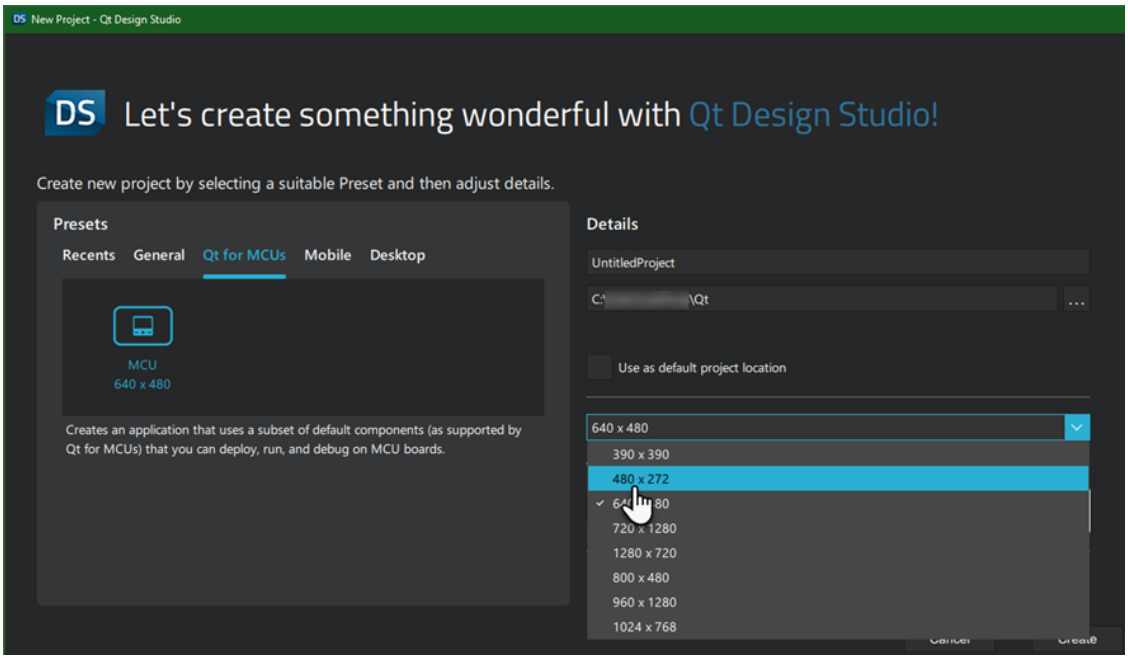
The following instructions guide you through the complete design process:

1. Create a new project

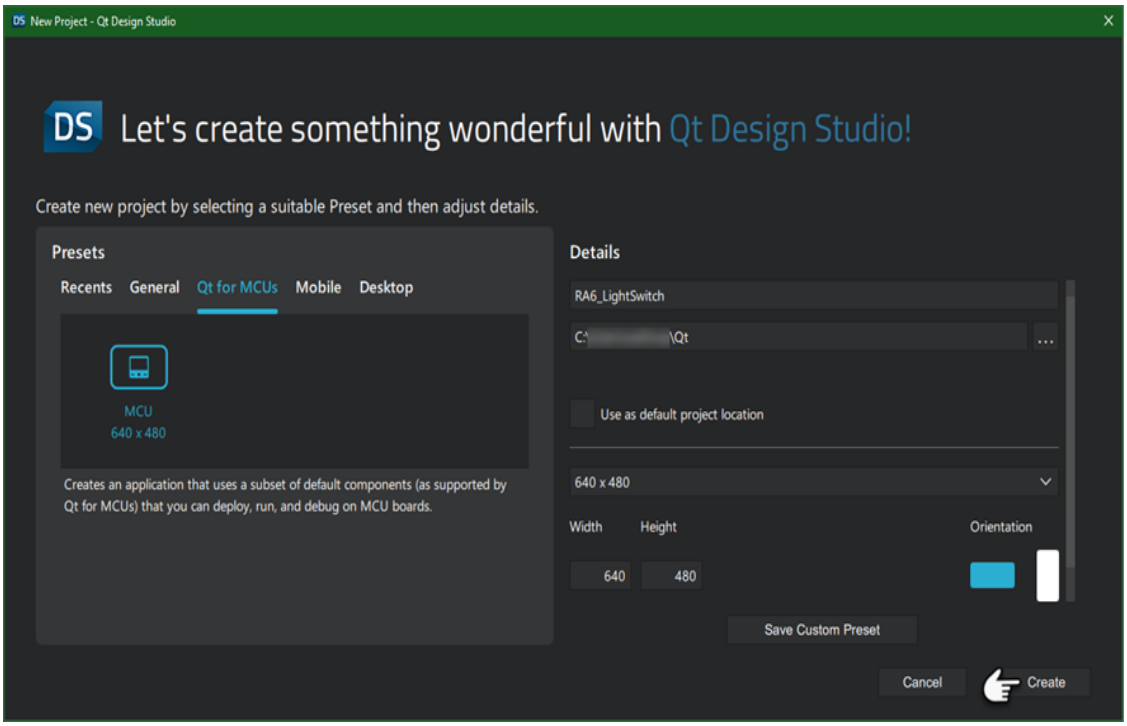
- Launch Qt Design Studio and click Create Project



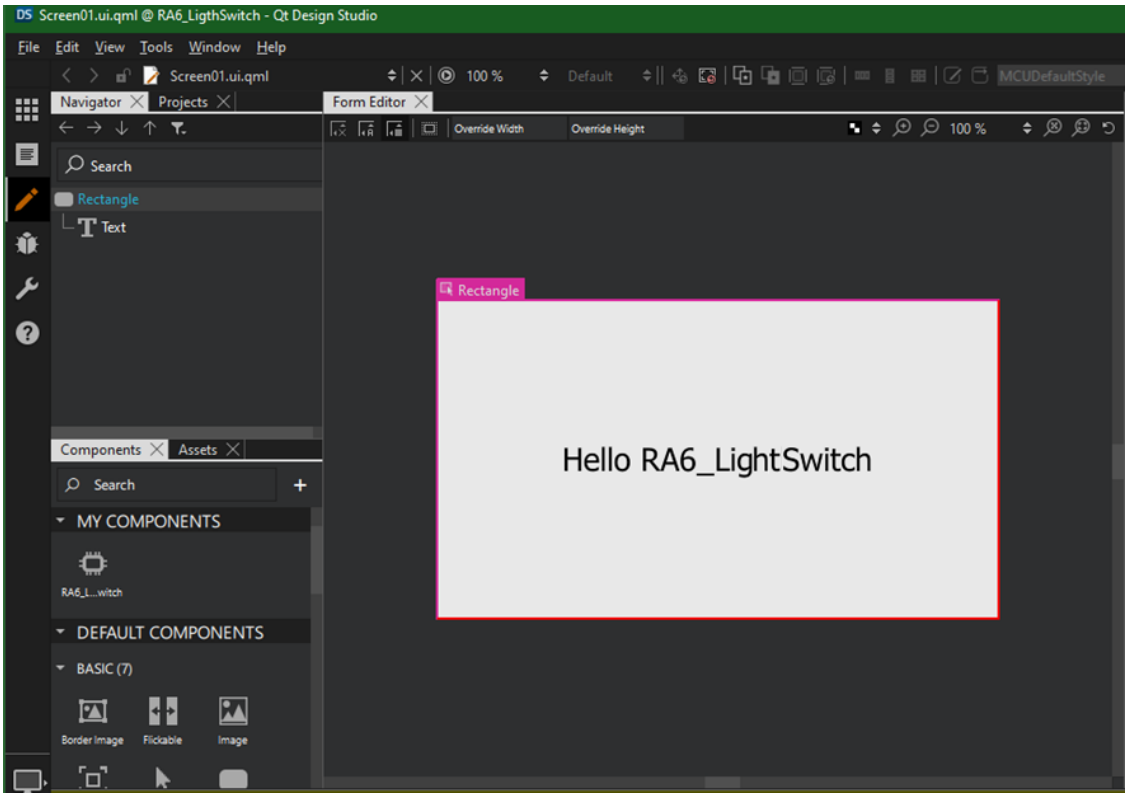
- Select the "Qt for MCUs" tab in the New Project wizard
- Select 480 x 272 from the resolution drop-down menu



- Name your project and click Create

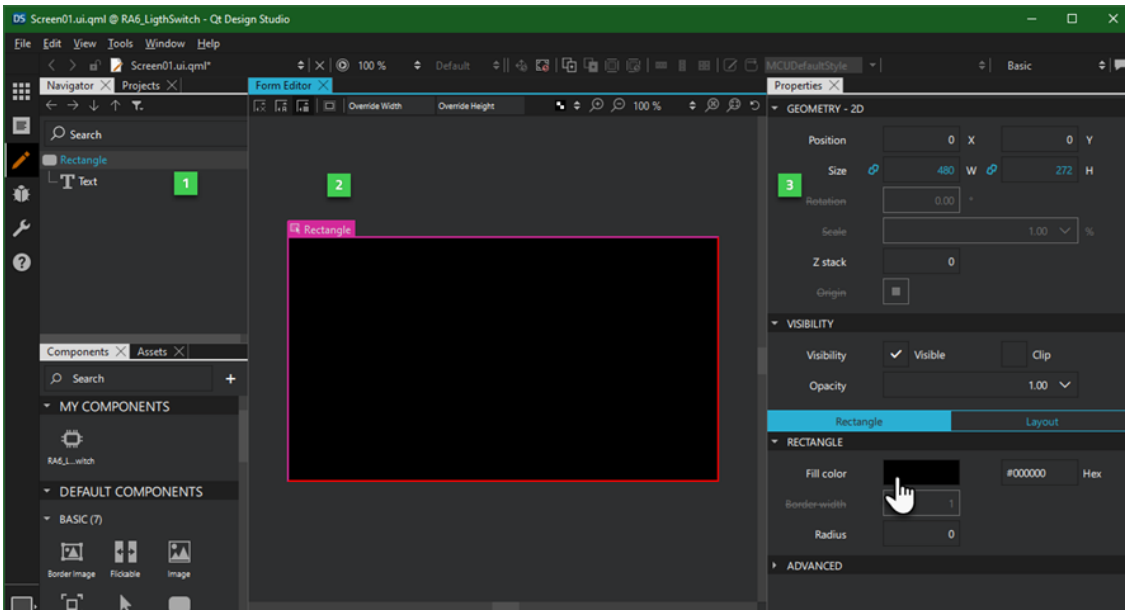


You should now see the Design Mode UI and the boilerplate project with a Rectangle and Text item.



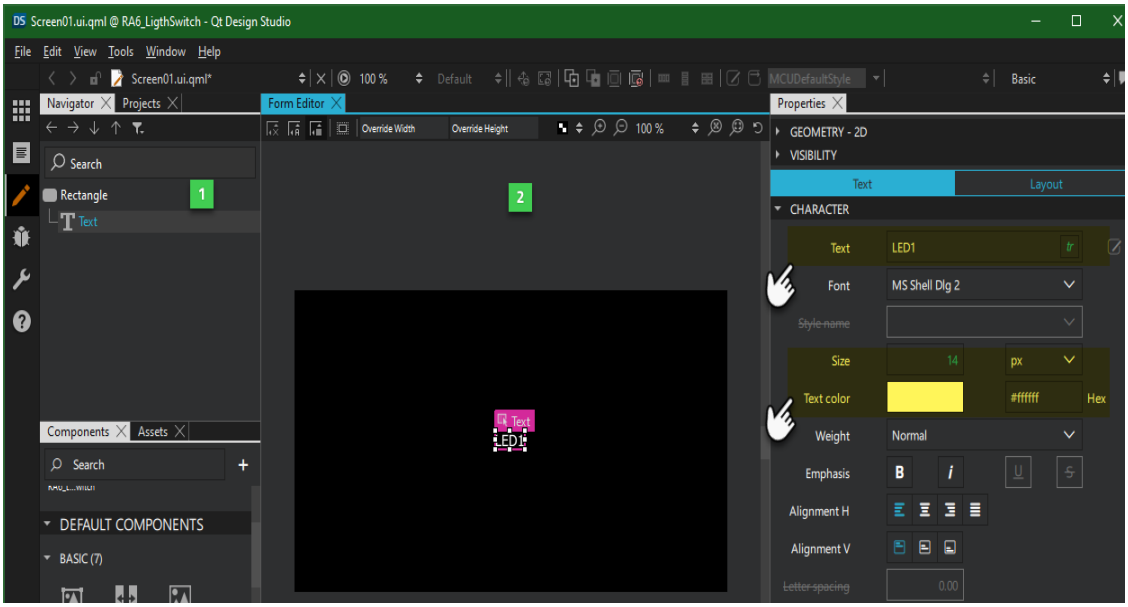
2. Select the Rectangle either in the Navigator 1 pane or Form Editor 2 to change its properties in the Properties 3 pane:

- Change the Fill color property to black either by using the color picker or entering Hex code (#000000) manually.



3. Select the Text item either in the Navigator 1 pane or Form Editor 2 to change its properties:

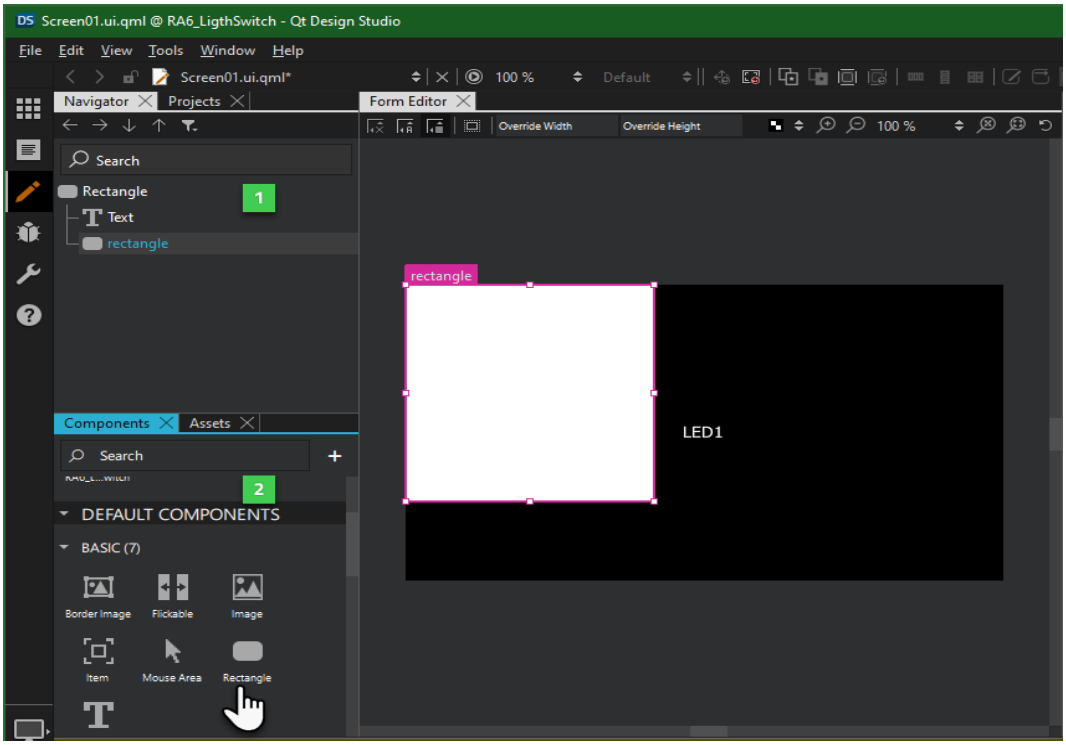
- Change Text Color to white either using the color picker or entering Hex code (#ffffff) manually.
- Change Units to px and Size to 14
- Change Text to LED 1



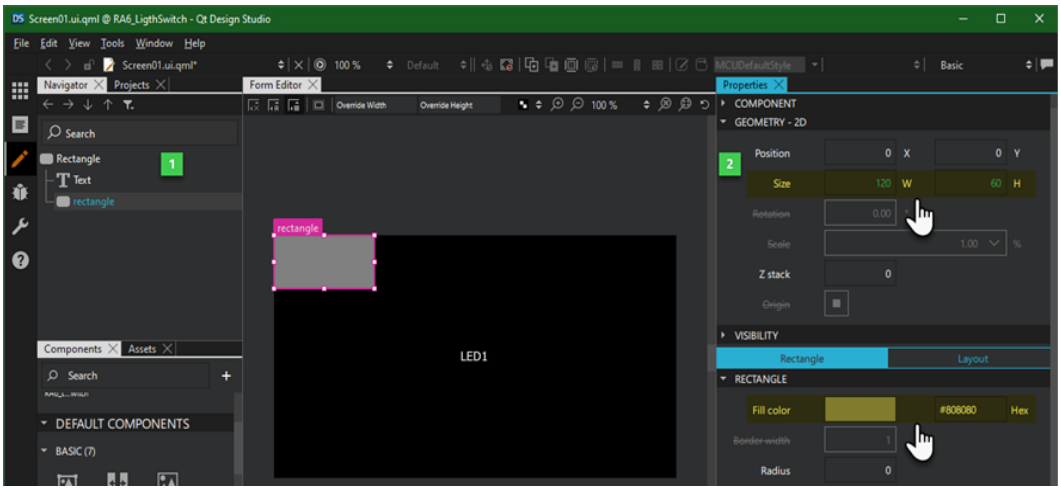
4. Add the button rectangle

- Find the Rectangle component in the Components 2 pane under the Basic components.

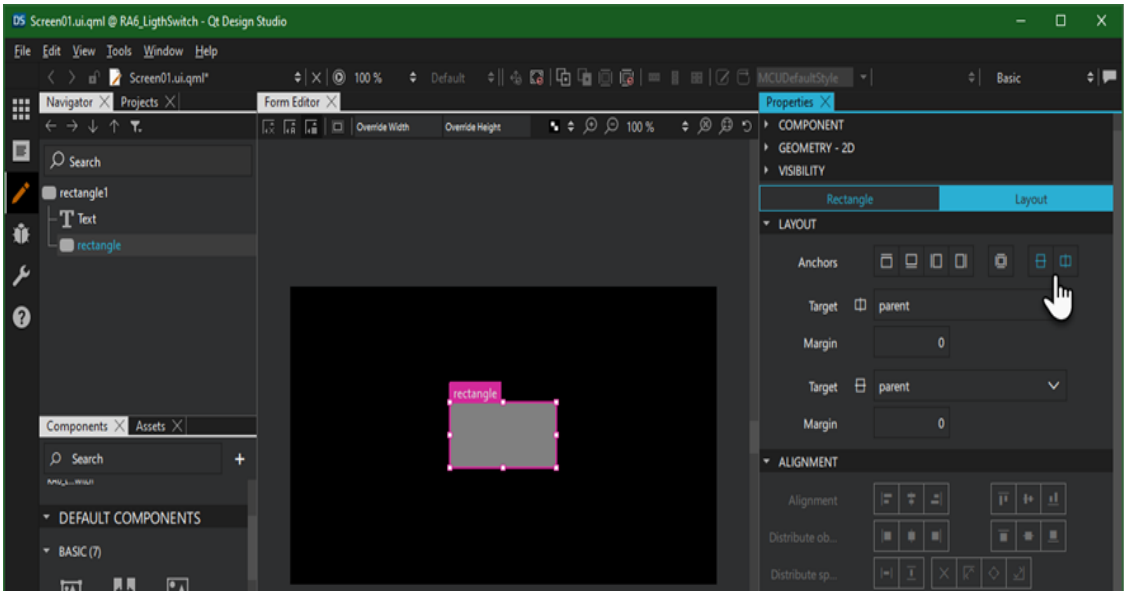
  Drag it onto the Rectangle in the Navigator 1 pane.



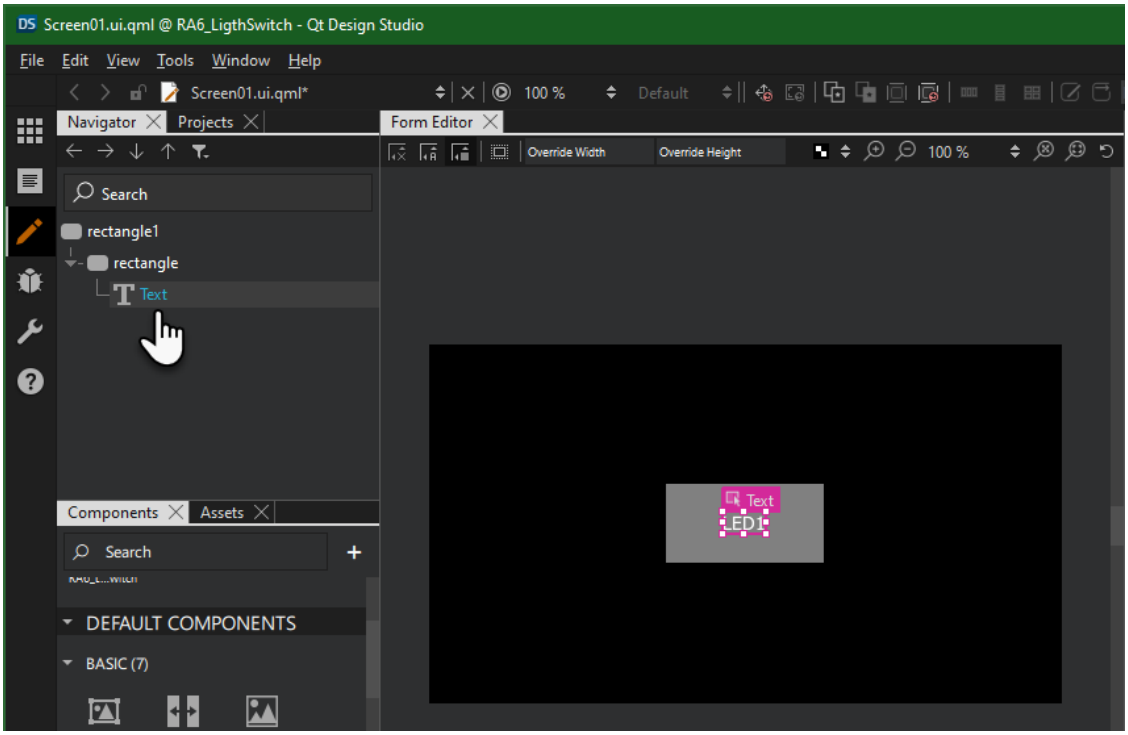
- Select the new rectangle in the Navigator 1 pane to change some of its properties in the Properties 2 pane:
  - Change its Fill color property to grey either using the color picker or entering the Hex code (#808080) manually.
  - Size to 120 W x 60 H



  - Switch to Layout section in the Properties pane. Anchor the rectangle to the center of its parent.

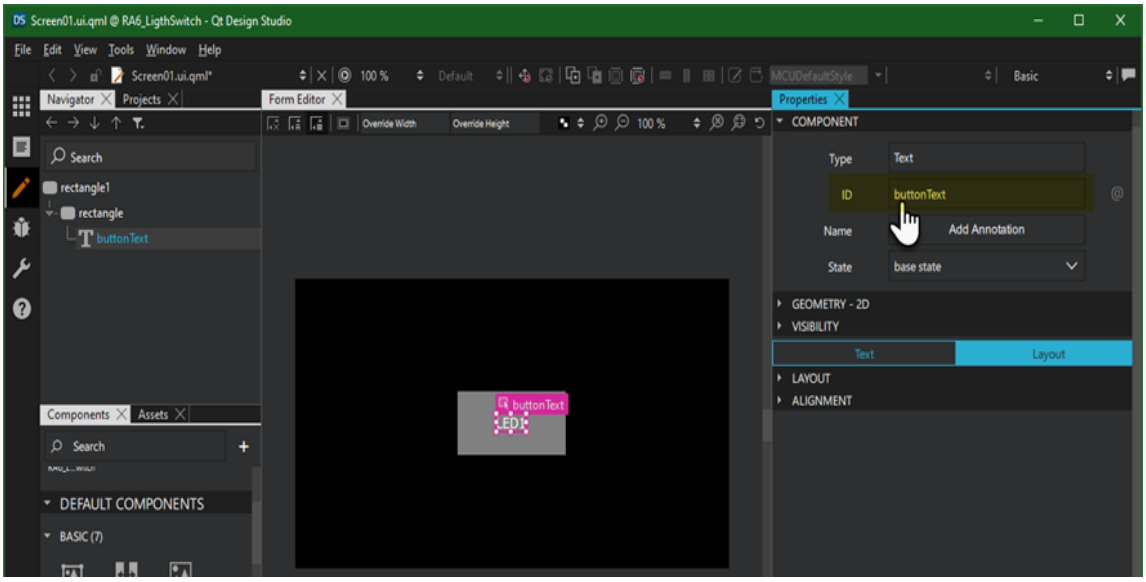


- Drag the existing Text component onto the new Rectangle.



5. Assign unique ID to each item:

- Name the Text item as `buttonText`:

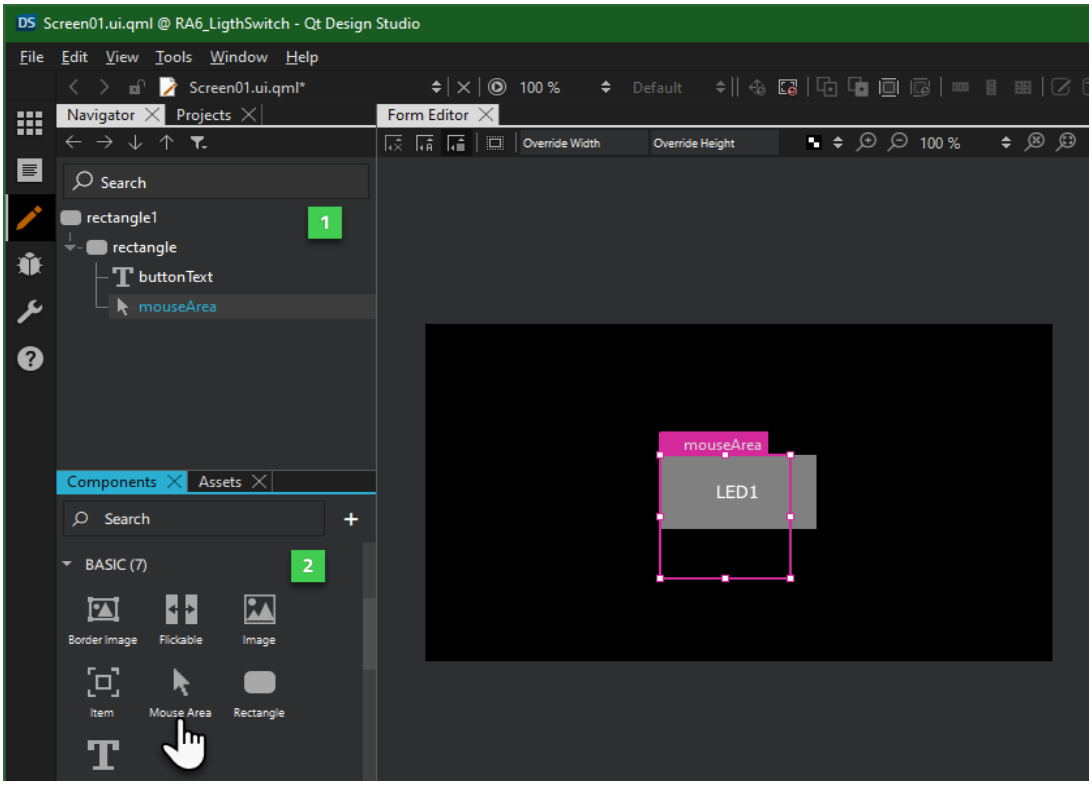


- Name the inner Rectangle item as `buttonRectangle`
- Name the parent Rectangle item as `backgroundRectangle`

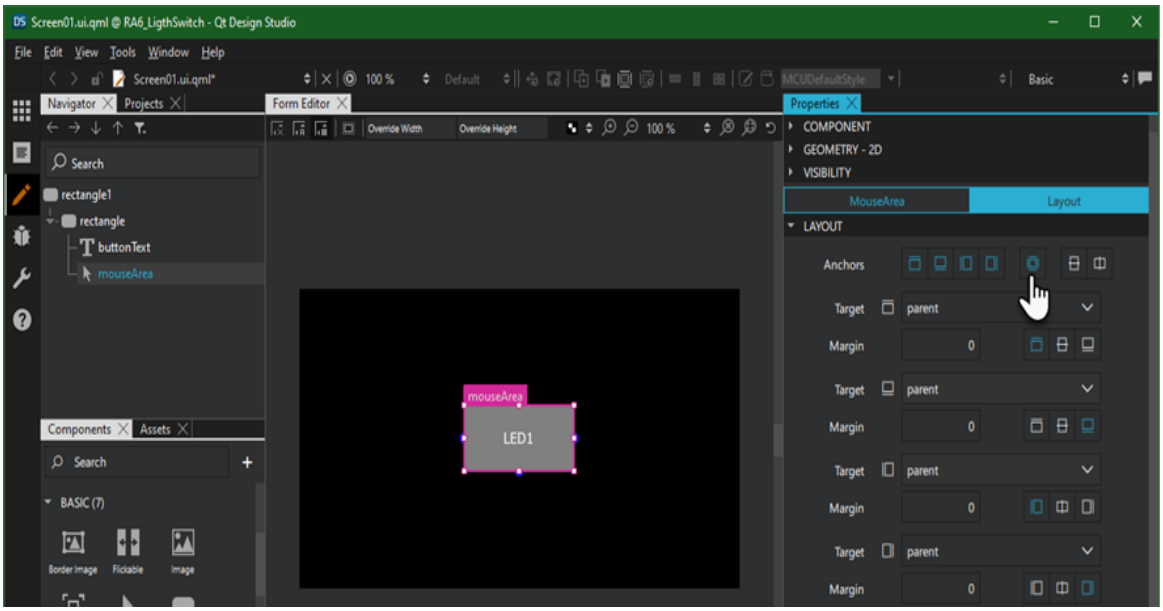
6. Add a MouseArea

- Find the MouseArea component in the Components 2 pane under the Basic components.

  Drag it onto the inner Rectangle in the Navigator 1 pane.

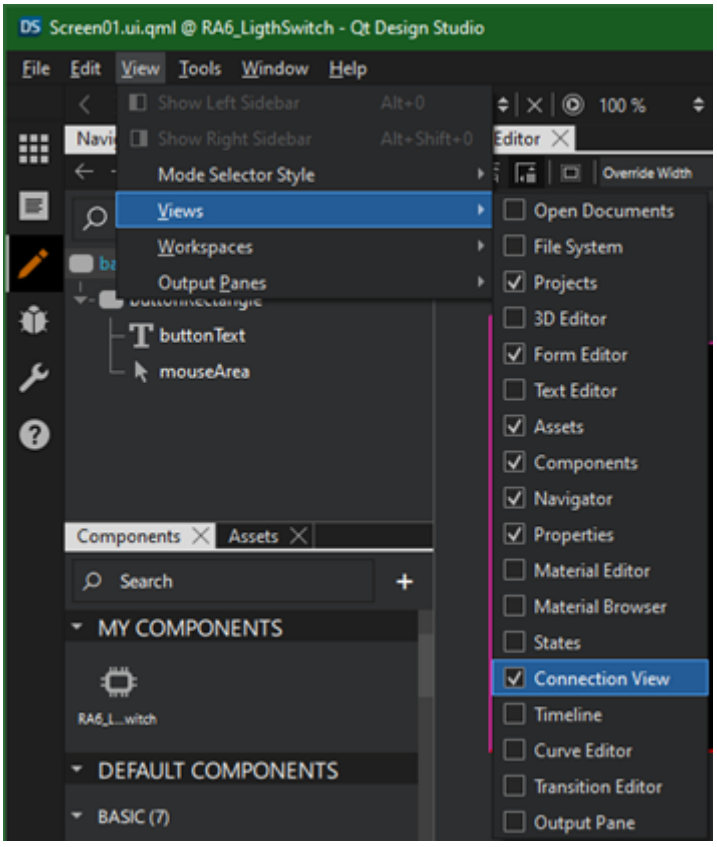


- Double-click on the MouseArea component to rename it as `buttonMouseArea`.
- Anchor it to span its parent, buttonRectangle.

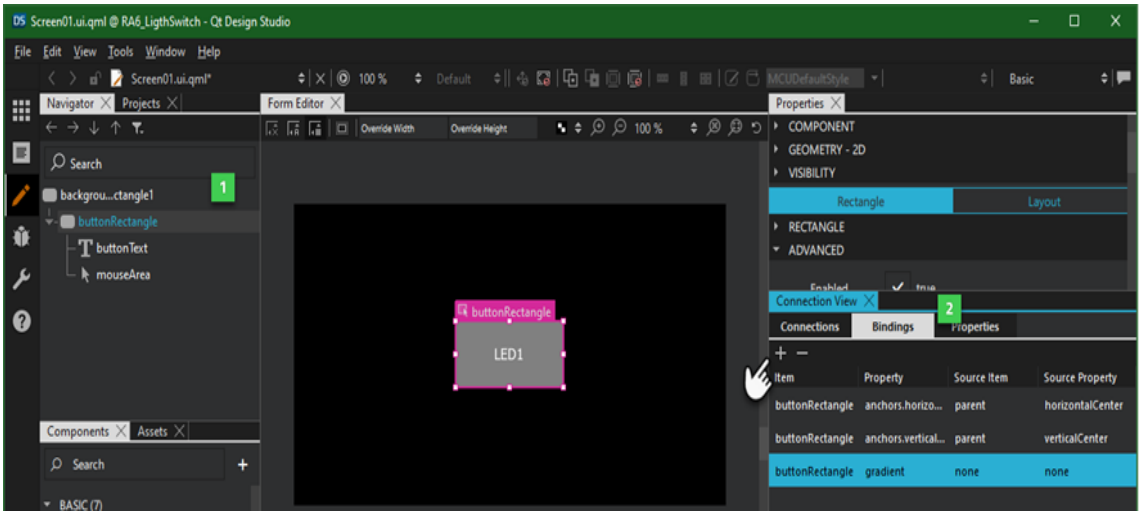


7. Bind the button's color property to the MouseArea's pressed property:

- Select buttonRectangle in the Navigation 1 pane
- Select Connection View from the View > Views menu.



- Switch to the Bindings tab in the Connection View 2.
- Click + to add new binding

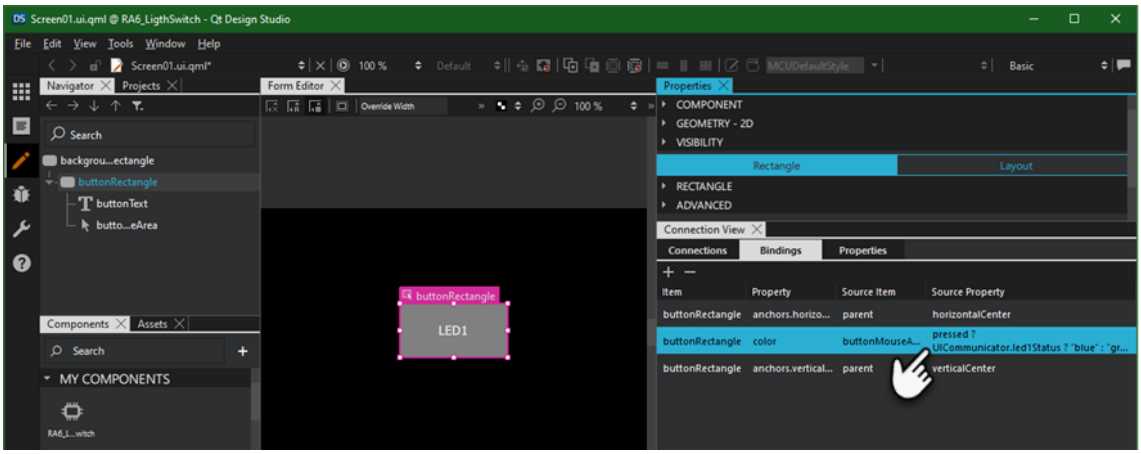


- Select color in the Property column for new binding
- Select mouseArea in the Source Item column
- Right-click on the new binding, click Open Binding Editor, enter the following expression, then click OK:

```
buttonMouseArea.pressed ?
    UICommunicator.led1Status ? "blue" : "grey" :
    UICommunicator.led1Status ? "darkblue" : "dimgrey"
```

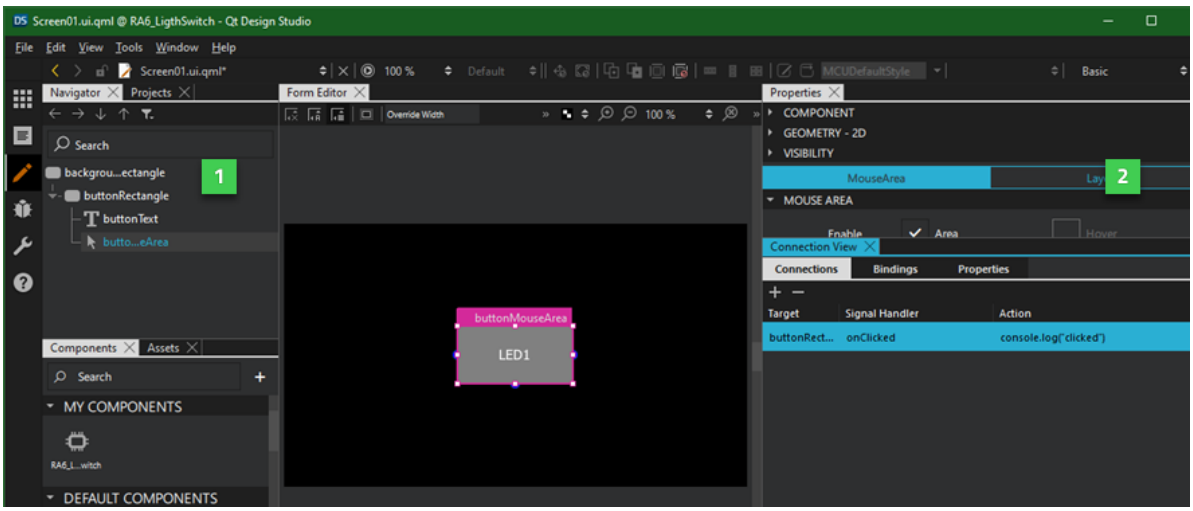
Note

UICommunicator is the C++ backend that you add in the next chapter.



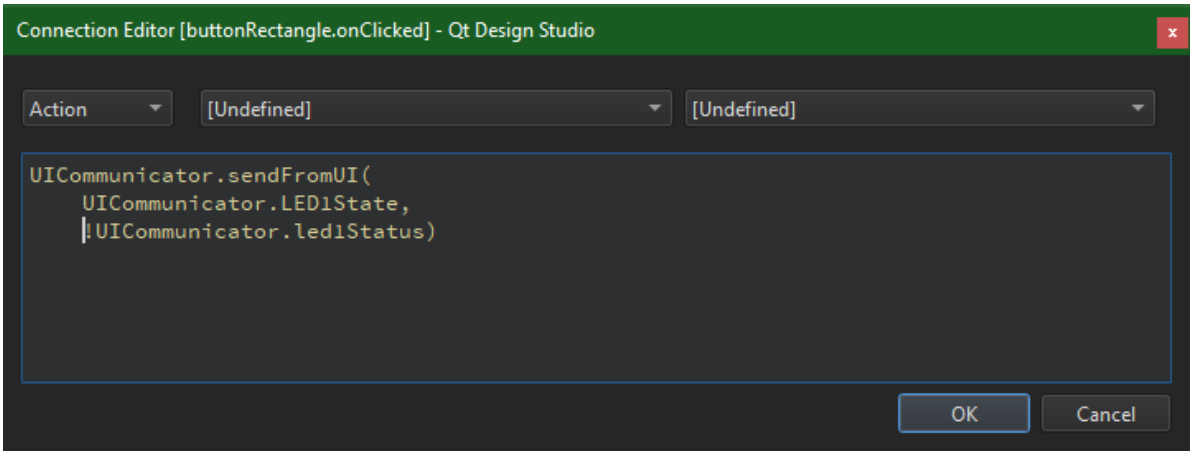
8. Add a connection for the MouseArea's onClicked handler:

- Select buttonMouseArea in the Navigation 1 pane
- Select the Connections tab in the Connection View 2 pane
- Click + to add a new connection



- Right-click on the new connection and choose Open Connection Editor
- Add the following code in the editor then click OK:

```
UICommunicator.sendFromUI(
    UICommunicator.LED1State,
    !UICommunicator.led1Status)
```

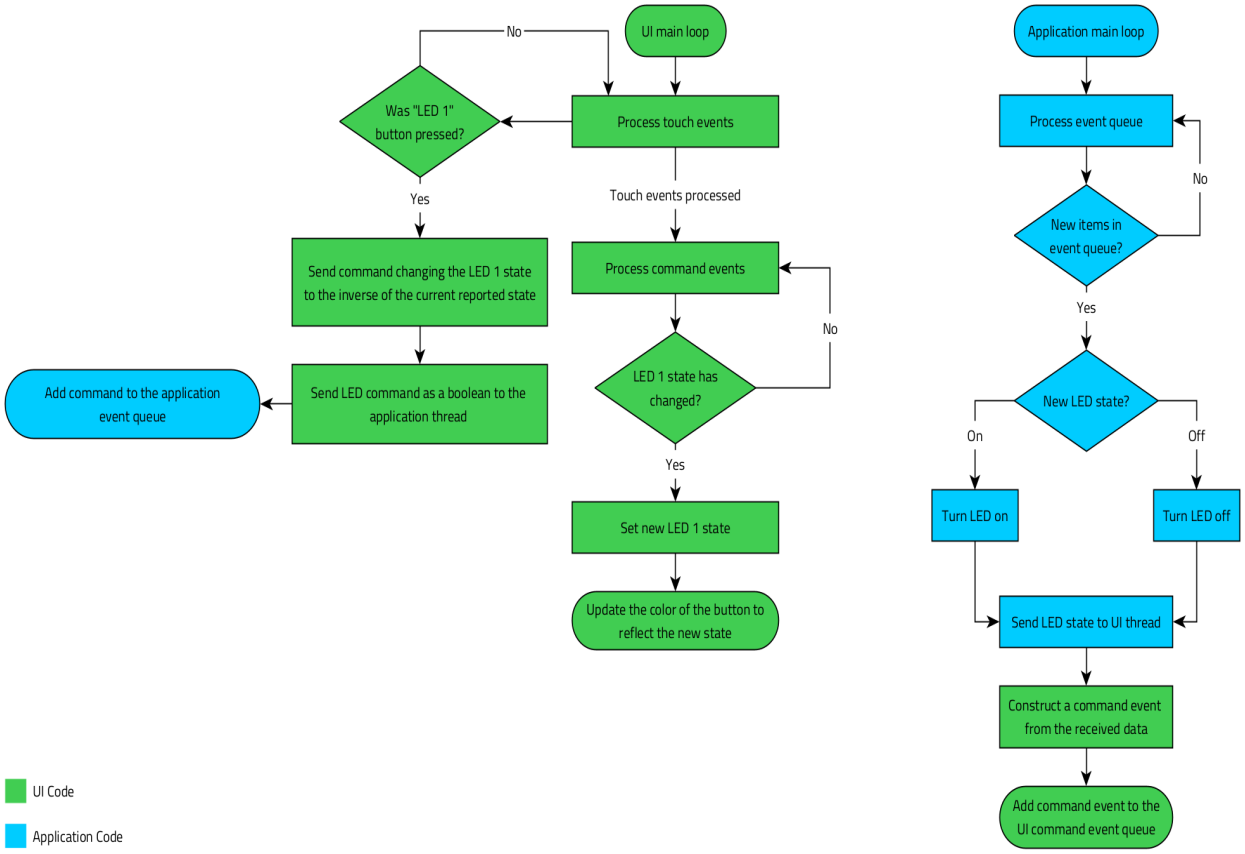


9. Save and close

- Select File > Save All
- Select File > Close Project <*your-project-name*>

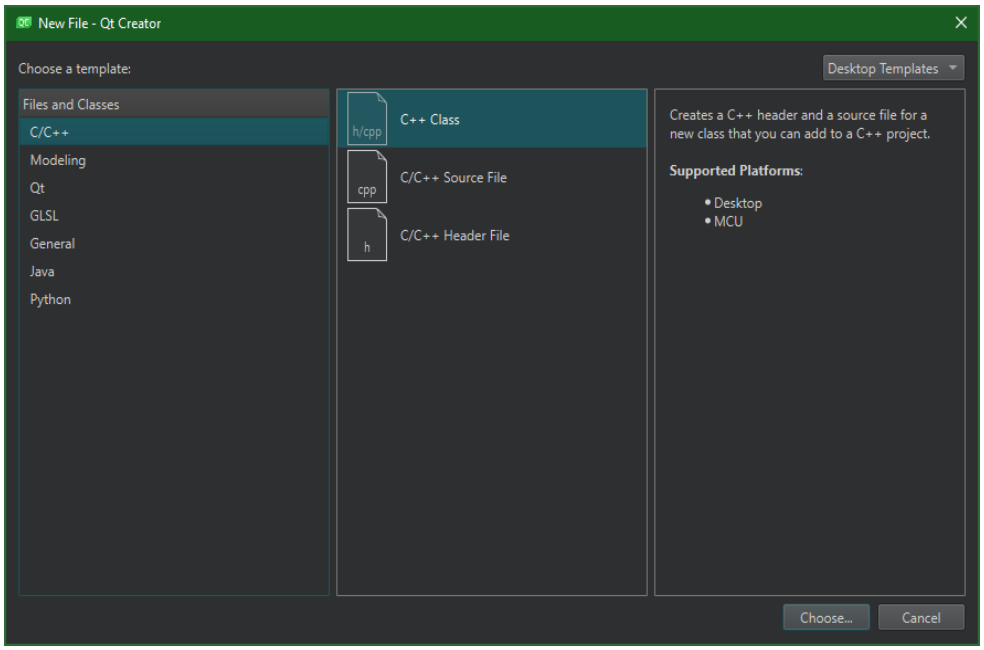
2. Create and Build the Application Backend

This chapter guides you through the steps to create and build the application's backend using Qt Creator, which you get as part of the Qt for MCUs installation. The backend enables the application's UI to communicate with the platform and get the required information from the hardware. In this case, the communicator gets the status of the on-board LED. The following diagram describes the workflow between the two components:

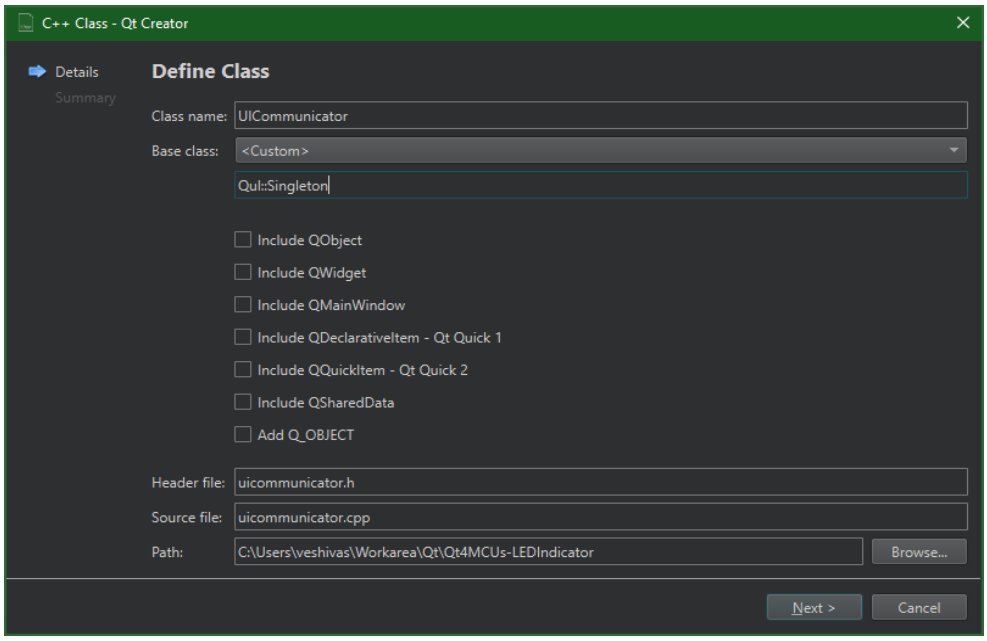


The following instructions guide you through the entire process:

1. Launch Qt Creator and open the project that you created with Qt Design Studio in the previous chapter:
   - Click File then Open File or Project.
   - Navigate to your project folder and double-click the CMakeLists.txt file.
   - In the Configure Project window that appears, select the kits Qt for MCUs 2.2 - Desktop 32bpp and Qt for MCUs 2.2 - EK-RA6M3G-BAREMETAL 16bpp (ARMGCC). Deselect any others, then click the Configure Project button near the bottom of the window.
2. Select File > New File to add a new C++ Class to the project. Name the new class `UICommunicator` and enter `Qul::Singleton` as its base class.



3. Click Next and choose the correct project (your project name) from the list before you click Finish.



5. Click Yes in the resulting pop-up window to copy the source file names and append them to the qul_add_target... line in CMakeLists.txt:

   `qul_add_target(<project-name> uicommunicator.h uicommunicator.cpp)`

   Then add the following line:
   `qul_target_generate_interfaces(<project-name> uicommunicator.h)`

6. Navigate to uicommunicator.h via the "Header Files" section in the Projects pane, or via the File System view (switch from “Projects” view at upper left). Edit the file to contain the following:

```
#ifndef UICOMMUNICATOR_H
#define UICOMMUNICATOR_H

#include <qul/singleton.h>
#include <qul/property.h>
#include <qul/eventqueue.h>
```

```
struct UICommunicator : public Qul::Singleton<UICommunicator>
{
    friend struct Qul::Singleton<UICommunicator>;

    enum Command { LED1State };

    Qul::Property<bool> led1Status;

    void sendFromUI(Command command, bool commandData);
    void receiveToUI(Command command, bool commandData);

private:
    UICommunicator();
    UICommunicator(const UICommunicator &);
    UICommunicator &operator=(const UICommunicator &);
};

struct CommandEvent
{
    UICommunicator::Command command;
    bool commandData;
};

class CommandEventQueue :
public Qul::EventQueue<struct CommandEvent, Qul::EventQueueOverrunPolicy_Discard,
 10>
{
public:
    void onEvent(const CommandEvent &commandEvent);
};

#endif // UICOMMUNICATOR_H
```

The header declares the UICommunicator struct, which inherits *Qul::Singleton*, making it safe to call from C++ code. For more information, refer to Singleton class reference.

The header also declares the Command enum that contains a list of commands, and the CommandEventQueue to manage the queue. The header also declares the led1Status property to indicate the status of the on-board blue LED. This property is exposed to the QML context, which uses it to determine the color of the button.

The UICommunicator class also contains the sendFromUI and receiveToUI functions to send and receive commands. In addition, the CommandEventQueue is used to communicate with the UI thread in a thread-safe way. Instead of calling receiveToUI from the application thread, the commands are added to the CommandEventQueue, which is then processed by the QUL thread to call receiveToUI.

7. Replace the code in uicommunicator.cpp with the following:

```
#include "uicommunicator.h"

extern void sendCommandToAppThread(bool led1Status);

UICommunicator::UICommunicator()
{
    led1Status.setValue(false);
}

void UICommunicator::sendFromUI(Command command, bool commandData)
{
    QUL_UNUSED(command)
```

```
    sendCommandToAppThread(commandData);
}

void UICommunicator::receiveToUI(Command command, bool commandData)
{
    switch (command) {
    case LED1State:
        led1Status.setValue(commandData);
        break;
    default:
        break;
    }
}

void CommandEventQueue::onEvent(const CommandEvent &commandEvent)
{
    UICommunicator::instance().receiveToUI(
      commandEvent.command, commandEvent.commandData);
}

static CommandEventQueue commandEventQueue;

void sendToUI(bool led1Data)
{
    CommandEvent commandEvent;
    commandEvent.command = UICommunicator::LED1State;
    commandEvent.commandData = led1Data;
    commandEventQueue.postEvent(commandEvent);
}
```

The extern symbol at the beginning should be moved to the e^2 studio project you will create in the next chapter. For now, it's declared here to ensure the UI project compiles.

The UICommunicator class sets led1Status to false. Its sendFromUI() member function sends a boolean value to the application thread. The receiveToUI() member function uses the command argument to determine if the property must be updated.

Next, the CommandEventQueue class overrides the onEvent() function. This function override calls receiveToUI() on the UICommunicator instance, with the command and commandData parameters.

In addition, create a static instance of CommandEventQueue, which is used by the sendToUI() function to post events. The function constructs a CommandEvent from the given boolean value, and adds it to the commandEventQueue for processing. The sendToUI() method is called from the application thread when the LED1's state changes.

8. *Optional*: if desired, you can now connect an RA6M3G board to your computer and use Qt Creator to build, flash and run the project in its current state. First, temporarily replace the extern at the top of uicommunicator.cpp with a no-op (empty) function, and save. Then, in the lower left hand corner, where you see the word "Debug", click it and select the RA6 kit in the submenu that appears. Click outside the submenu to exit it, then click the green arrow immediately below the word "Debug". The app should build and flash to the device, and run there automatically. Be sure to replace the original extern declaration before continuing.

9. Edit CMakeLists.txt to append the STATIC_LIBRARY argument to the qul_add_target function call. The qul_add_target line should look like this after the edit:

```
qul_add_target(<project-name> STATIC_LIBRARY uicommunicator.h uicommunicator.cpp)
```

10. Build your project against the Qt for MCUs 2.2 - EK-RA6M3G-BAREMETAL 16bpp (ARMGCC) kit.

After a successful build, you should see the static library and qul_run.h C++ header in the build directory.

Note

Qt Creator uses “shadow builds” by default. This is an out-of-source build where the build artifacts live in a directory that is parallel to the source directory. The build directory name depends on the project name, kit, and the build type (debug or release). For example, a debug build of project Foo against the Qt for MCUs 2.2 - Desktop kit, should have build-Foo-Qt_for_MCUs_2_2_Desktop_32bpp_Debug as its build directory.

3. Create a Renesas e^2 studio Project

This chapter provides you step-by-step instructions to create a Renesas e^2 studio project, and integrate the static library you built in the last chapter.

The following instructions guide you through the complete process:

1. Launch e^2 studio and create a new project for RA6M3G using File > New > Renesas C/C++ Project > Renesas RA. Select C++ as the project language. Click Next and select Executable and FreeRTOS.
2. Click Next and select FreeRTOS - Minimal – Static Allocation as the project template. Follow the wizard to create the project.
3. Configure your FSP Stacks in the Stacks tab of the FSP Configuration editor, as described in the next chapter.
4. Import all header and source files from the included platform sources archive. As a result, you should have a source directory named platform in your project, containing the sources for *Qt Quick Ultralite RA6M3G FreeRTOS* platform port.
5. Open C/C++ Project Settings and make the following changes:
   - Add the platform folder to the project's source locations. Right-click on the project in the Project Explorer and choose Properties, then C/C++ General > Paths and Symbols > Source Location.
   - Add *<your-Qt-Creator-build-folder>*/*<Release|Debug|MinSizeRel>* and *<Qt-install-dir>*/QtMCUs/*<QUL-version>*/include to the C++ include directories list under C/C++ General > Paths and Symbols > Includes. Check "Add to all configurations" and "Add to all languages"
   - Add the following libraries to the C/C++ Build > Settings > Tool Settings > GNU Arm Cross C++ Linker > Libraries list:
     - YOUR_GUI_APP_DIR/[Release/Debug/MinSizeRel]/libYourGUIApp.a
     - Add the following from QT_INSTALL_DIR/QtMCUs/[QUL_VERSION]/lib:
       - libMonotypeUnicode_cortex-m4-hf-fpv4-sp-d16_Windows_armgcc_MinSizeRel.a
       - libQulMonotypeUnicodeEngineShaperDisabled_cortex-m4-hf-fpv4-sp-d16_Windows_armgcc_MinSizeRel.a
       - libQulPNGDecoderLodePNG_cortex-m4-hf-fpv4-sp-d16_Windows_armgcc_MinSizeRel.a
       - libQulCore_cortex-m4-hf-fpv4-sp-d16_Windows_armgcc_MinSizeRel.a
     - Add QT_INSTALL_DIR/QtMCUs/[QUL_VERSION]/lib to the Library Search Path
   - Set the C++ language standard to GNU ISO 2014 C++ under C/C++ Build > Settings > Tool Settings > Gnu Arm Cross C++ Compiler > Optimization. Check the Do not use exceptions, Do not use RTTI, and Do not use thread-safe statics options.

     Note

     These settings should be applied to all project configurations.
   - Add the following preprocessor definitions under C/C++ Build > Settings > Tool Settings > GNU Arm Cross C++ Compiler:

```
QUL_PLATFORM_DEFAULT_TEXT_CACHE_ENABLED=0
QUL_PLATFORM_DEFAULT_NUM_FRAMES_TO_PRESERVE_ASSETS=0
QUL_PLATFORM_DEFAULT_TEXT_CACHE_SIZE=24*1024
```

```
QUL_PLATFORM_REQUIRED_PIXEL_WIDTH_ALIGNMENT=1
QUL_PLATFORM_REQUIRED_IMAGE_ALIGNMENT=1
QUL_PLATFORM_DEFAULT_RESOURCE_PIXEL_FORMAT_ALPHA=ARGB8888
QUL_PLATFORM_DEFAULT_RESOURCE_PIXEL_FORMAT_OPAQUE=RGB565
QUL_COLOR_DEPTH=16
```

– Add `ucHeap=__HeapBase` preprocessor definition under C/C++ Build > Settings > Tool Settings > GNU Arm Cross C Compiler.

Note

This makes FreeRTOS use the heap that is usually reserved for std malloc. Ensure that your project does not use std mallocs or relocate FreeRTOS heap for other purposes. The current RA6 FreeRTOS platform port uses FreeRTOS pvPortMalloc(), so it needs a sizable portion of RAM to work. You can change this behavior by overriding the functions in platform/mem-freertos.cpp.

6. Edit script/fsp.ld to insert the following code before the last curly brace in the file:

```
QulModuleResourceData :
{
. = ALIGN(4);
__qspi_flash_start__ = .;
*(QulModuleResourceData)
} > QSPI_FLASH

QulFontResourceData :
{
. = ALIGN(4);
*(QulFontResourceData)
} > QSPI_FLASH

QulResourceData :
{
. = ALIGN(4);
*(QulResourceData) . = ALIGN(4);
__qspi_flash_end__ = .;
} > QSPI_FLASH
```

7. Edit src/hal_entry.cpp to insert the following code in the R_BSP_WarmStart() function definition. Note that this needs to be inside the BSP_WARM_START_POST_C if block, right after R_IOPORT_Open().

```
R_ICU_ExternalIrqOpen(&g_S1_irq_ctrl, &g_S1_irq_cfg);
R_ICU_ExternalIrqEnable(&g_S1_irq_ctrl);
```

This will enable the S1 button on the board.

Edit qul_thread_entry.cpp and replace the existing code with the following:

```
#include "qul_thread.h"
#include "qul_run.h"

void qul_thread_entry(void *pvParameters)
{
   FSP_PARAMETER_NOT_USED (pvParameters);

   qul_run();
}

static bool ledEvent = false;
void sendCommandToAppThread(bool led1Status)
```

```
{
   ledEvent = led1Status;
   xQueueSend(g_app_queue, &ledEvent, 0);
}
```

The qul_thread_entry() function ensures that it gives the thread to *Qt Quick Ultralite*. The sendCommandToAppThread() function is called from the UI code, to get the requested led1Status and post it to g_app_queue.

Edit app_thread_entry.cpp to replace the code with the following:

```
#include "app_thread.h"

extern void sendToUI(bool led1Data);

void app_thread_entry(void *pvParameters)
{
   FSP_PARAMETER_NOT_USED(pvParameters);
   bool led1State;
   while(true)
   {
      /* Wait for an event from the user interface */
      if (pdTRUE == xQueueReceive(g_app_queue, (void*) &led1State,
portMAX_DELAY))
      {
         bsp_io_level_t level;
         level = led1State ? BSP_IO_LEVEL_HIGH : BSP_IO_LEVEL_LOW;
         R_BSP_PinAccessEnable();
         R_BSP_PinWrite(BSP_IO_PORT_04_PIN_03, level); /* blue LED on EK-RA6M3 */
         R_BSP_PinAccessDisable();
         sendToUI(led1State);
      }
   }
}

static bool led1Level = false;
extern "C" void s1_irq_callback(external_irq_callback_args_t *p_args)
{
   FSP_PARAMETER_NOT_USED(p_args);
   R_BSP_PinAccessEnable();
   uint32_t level = R_BSP_PinRead(BSP_IO_PORT_04_PIN_03); /* blue LED on EK-RA6M3
*/
   R_BSP_PinAccessDisable();
   led1Level = !(level & 0x01);
   xQueueSendFromISR(g_app_queue, &led1Level, NULL);
}
```

Notice that the first line declares sendToUI extern symbol. This was earlier defined in the application backend code in the previous chapter.

Next, the app_thread_entry() function uses a simple while loop, waiting for an item on g_app_queue. When it finds an item (a boolean value) on the queue, it toggles (on/off) the blue LED1 based on the item value. Finally, it sends the change notification to the UI thread.

The last function is the callback for s1 interrupt defined in the FSP configuration. It checks the LED1 status and adds the inverse value to the g_app_queue, to toggle the LED1 depending on its previous state.

Your application is now ready. Build and flash it to the RA6M3G board to test that everything works as intended. Next, you can try experimenting with the code. For example, an interrupt for the S2 button is already defined but it is not used yet. Implement a callback for S2 to toggle LED2 and update the LED's status in the UI.

4. FSP Configuration for RA6M3G

To get Qt Quick Ultralite working, the FSP needs quite extensive configuration. Use FSP editor in e^2 studio to add the following FSP stacks and threads:

ADC Driver r_adc

Property	Value
General > Name	g_adc0
Input > Channel Scan Mask > Channel 2	Selected
Pins > AN02	P002

D/AVE 2D Port Interface r_drw

Property	Value
Common > Allow Indirect Mode	Enabled
Common > Memory Allocation	Default
Module > D2 Device Handler Name	d2_handle0
Module > DRW Interrupt Priority	Priority 2

Display Driver r_glcdc

Property	Value
Module >General > Name	g_display0
Interrupts > Callback Function	glcdc_callback
Interrupts > Line Detect Interrupt Priority	Priority 2
Input > Graphics Layer 1 > Framebuffer > Number of framebuffers	1

External IRQ Driver r_icu

Property	Value
Module > Name	g_touch_irq
Module > Channel	0
Module > Trigger	Falling
Module > Digital Filtering	Enabled
Module > Digital Filtering Sample Clock	PCLK / 64
Module > Callback	touch_irq_cb
Module > Pin Interrupt Priority	Priority 5
Pins > IRQ00	P206

Note

Pin P206 must be set to IRQ mode to ensure correct functionality, and its Pull up property must be set to input pull-up.

I²C Master Driver r_iirc_master

Property	Value
Module > Name	g_i2c_touch
Module > Channel	2
Module > Rate	Fast-mode
Module > Rise Time (ns)	120
Module > Fall Time (ns)	120
Module > Duty Cycle (%)	50
Module > Slave Address	0x38
Module > Address Mode	7-bit
Module > Timeout Mode	Short Mode
Module > Callback	touch_i2c_callback
Module > Interrupt Priority Level	Priority 6
Pins > SDA	P511
Pins > SCL	P512

QSPI Driver r_qspi

Property	Value
Module > General > Name	g_qspi0
Module > Bus Timing > Minimum QSSL Deselect	8 QSPCLK

Timer Driver r_gpt

Property	Value
Common > Pin Output Support	Enabled
Module > General > Name	g_timer_PWM
Module > General > Channel	7
Module > General > Mode	PWM
Module > General > Period	10
Module > General > Period Unit	Milliseconds
Module > Output > Duty Cycle Percent	75
Module > Output > GTIOCA Output Enabled	True
Pins > GTIOCA	P603

Note

You must set Operation Mode of the GPT Timer to GTIOCA or GTIOCB when setting the GTIOCA pin, and pin P603 must be set to Peripheral mode.

UART Driver r_sci_uart

Property	Value
Module > General > Name	g_uart0
Module > General > Channel	0
Module > General > Data Bits	8bits
Module > General > Parity	None
Module > General > Stop Bits	1bit
Module > Interrupts > Callback	user_uart_callback
Pins > TXD_MOSI	P411
Pins > RXD_MISO	P410

Note

Set the Operation Mode to Asynchronous UART while assigning the pins for the UART. In addition, SPI0 must be disabled, or different pins must be assigned to use the P410 and P411 pins for UART.

S1 IRQ Driver r_icu

Property	Value
Module > Name	g_S1_irq
Module > Channel	13
Module > Trigger	Falling
Module > Digital Filtering	Enabled
Module > Digital Filtering Sample Clock	PCLK / 64
Module > Callback	s1_irq_callback
Module > Pin Interrupt Priority	Priority 12
Pins > IRQ13	P009

S2 IRQ Driver r_icu

Property	Value
Module > Name	g_S2_irq
Module > Channel	12
Module > Trigger	Falling
Module > Digital Filtering	Enabled
Module > Digital Filtering Sample Clock	PCLK / 64
Module > Callback	s2_irq_callback
Module > Pin Interrupt Priority	Priority 12
Pins > IRQ12	P008

Next, you will configure FreeRTOS for the project. For the LED indicator application, using single heap for everything should be sufficient.

Create a new thread in the FSP configuration tool. In the thread's properties, set the following Common properties:

Property	Value
Memory Allocation > Support Static Allocation	Enabled
Memory Allocation > Support Dynamic Allocation	Enabled
Memory Allocation > Total Heap Size	153600
Memory Allocation > Application Allocated Heap	Enabled

Set the following Thread properties:

Property	Value
Symbol	qul_thread
Name	QulThread
Stack Size (Bytes)	32768
Priority	4
Thread Context	NULL
Memory Allocation	Dynamic
Allocate Secure Context	Enable

Next, for HAL/Common add FreeRTOS Heap 4 stack. Create another thread and set the following properties:

Property	Value
Symbol	app_thread
Name	AppThread
Stack Size (Bytes)	1024
Priority	4
Thread Context	NULL
Memory Allocation	Dynamic
Allocate Secure Context	Enable

Next, create a new queue and set the following properties for it:

Property	Value
Symbol	g_app_queue
Item Size (Bytes)	1
Queue Length (Items)	20
Memory Allocation	Dynamic

Switch to the BSP tab in the FSP editor and click Generate project content to complete the process.

5. References

The following links provide some useful references that you could use to get more insight into Qt for MCUs.

- Managing Resources
  - https://doc.qt.io/QtForMCUs-2.2/qtul-resources.html
- Integrating C++ and QML
  - https://doc.qt.io/QtForMCUs-2.2/qtul-integratecppqml.html
- Entry point to Qt Quick Ultralite application
  - https://doc.qt.io/QtForMCUs-2.2/qtul-define-custom-entry-point.html
- Building a static library
  - https://doc.qt.io/QtForMCUs-2.2/qtul-building-application-as-a-static-library.html
- CMake Manual
  - https://doc.qt.io/QtForMCUs-2.2/qtul-cmake-manual.html

Website and Support

Visit the following URLs to learn about key elements of the RA family, download components and related documentation, and get support.

RA Product Information	www.renesas.com/ra
RA Product Support Forum	www.renesas.com/ra/forum
RA Flexible Software Package	www.renesas.com/FSP
Renesas Support	www.renesas.com/support
Qt Design Studio Support	https://doc.qt.io/qtdesignstudio/
Qt Support	https://www.qt.io/contact-us/technical-issue

Revision History

Rev.	Date	Summary
1.00	October 5, 2022	First release document.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.

(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

Renesas
TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

The Qt Company Oy
Miestentie 7
02150 Espoo, Finland
www.qt.io

Trademarks

Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/,
https://www.qt.io/contact-us