F&S Introduction to QT

Debugging an Application

Version 1.12
(02.05.2025)

About This Document

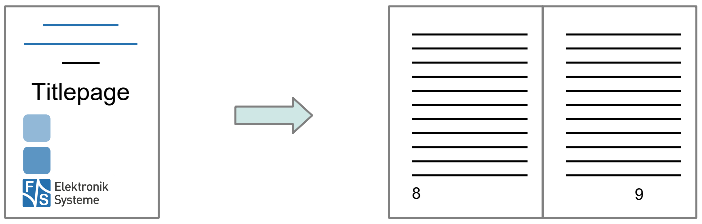
This document describes how to debugging remote device using QT Creator Automatic Remote Debugging Launcher under Linux. The software is configured for architectures fsimx6/fsimx6sx/fsimx6ul/fsimx7ulp/fsimx8mm from F&S under Linux/Buildroot.

Remark

The version number on the title page of this document is the version of the document. It is not related to the version number of any software release. The latest version of this document can always be found at http://www.fs-net.de.

How to Print This Document

This document is designed to be printed double-sided (front and back) on A4 paper. If you want to read it with a PDF reader program, you should use a two-page layout where the title page is an extra single page. The settings are correct if the page numbers are at the outside of the pages, even pages on the left and odd pages on the right side. If it is reversed, then the title page is handled wrongly and is part of the first double-page instead of a single page.



Typographical Conventions

We use different fonts and highlighting to emphasize the context of special terms:

`File names`

Menu entries

```
Board input/output
```

`Program code`

```
PC input/output
```

```
Listings
```

```
Generic input/output
```

`Variables`

History

Date	V	Platform	A,M,R	Chapter	Description	Au
2018-04-27	1.0	*	A	ALL	Create Documentation	PJ
2018-06-08	1.1	*	M	2.1, 5.2	Improve setup	PJ
2018-06-14	1.2	*	M	3	Update figure 12, it is necessary to use cmake instead of qmake	PJ
2020-05-04	1.3	*	M M M	2 3 All	Improve some setences Update figures for new QT Creator, improve some sentences Improve layout	PJ
2021-03-11	1.4	*	M	3, 5.2 5.3	Add GDB path for 64 bit CPUs Add target path for Yocto Add backend support for debugging	PJ
2022-07-19	1.5	*	M	2, 3, 5.1,5.2	Add Yocto Paths Updated some sentences	TG
2022-07-26	1.6	*	M	3	Changed Toolchain distro	TG
2022-09-19	1.7	*	A,M	1.2, 3,	Add Paths for i.MX6	TG
2022-11-09	1.8	*	M	1.1	Added installation for QT Creator 4.15.2 for Fedora 30	TG
2023-01-10	1.9	*	M	1, 2.2, 3	Updated figures for new QT Creator Added PicoCORE in Introduction	TG
2023-06-09	1.10	*	M	1.1,2.1,3	Add information about environment-setup Add information about sftp Fix missing "/" in command	PG
2023-09-28	1.11	*	A,M	1, 1.1, 1.2, 3, 4	Add general support for QT6 Add colored separation of displayed steps for QT5 and QT6 + Buildroot and Yocto separation Add paths for QT6	TG
2025-05-02	1.12	*	A,M	3, 6	Build Improvements and CMAKE Build instructions	SC

V	Version
A,M,R	Added, Modified, Removed
Au	Author

Table of Contents

1 Introduction

F&S offers a whole variety of Systems on Module (SOM) and Single Board Computers (SBC). There are different board families that are named NetDCU, PicoMOD, PicoCOM, PicoCore, armStone, QBliss, and efus.

Linux is available for all of these platforms. Also QT is available for all of these platforms. Linux offers a tool called Qt Creator which is a cross-platform C++, JavaScript and QML integrated development environment which is part of the SDK for the Qt GUI application development framework. It includes a visual debugger and an integrated GUI layout and forms designer.

This document describes how to debug an application on a remote device using the automatic remote debugging launcher plug-in.

This document describes both the steps for Qt5 and Qt6.

The Qt5 steps will be displayed like the following:

```
…example of paths or code.
```

And the Qt6 steps like this:

```
…example of paths or code.
```

If the steps are not displayed separately, it means that the steps work for both Qt5 and Qt6.

1.1 Install Qt Creator on Fedora

Install Qt Creator on your development host with your Linux Distribution specific package manager system. In our development machine, which you can download from our home-page, QT Creator is already installed. On Fedora up to version 27:

```
sudo yum install qt-creator
```

On Fedora 30 we will need to install a new version of QT Creator:

```
wget https://download.qt.io/official_releases/qtcreator/4.15/4.15.2/qt-
creator-opensource-linux-x86_64-4.15.2.run
```

Next we will need to give the file execute permissions:

```
chmod +x qt-creator-opensource-linux-x86_64-4.15.2.run
```

Now execute the file and follow the Installers instructions:

```
./qt-creator-opensource-linux-x86_64-4.15.2.run
```

Finally you want to create a link to the new QT Creator. Go into the directory that has been created for the new QT Creator and execute the following command:

```
sudo ln –sf /bin/qtcreator /usr/bin/qtcreator
```

Then start Qt Creator:

```
qtcreator &
```

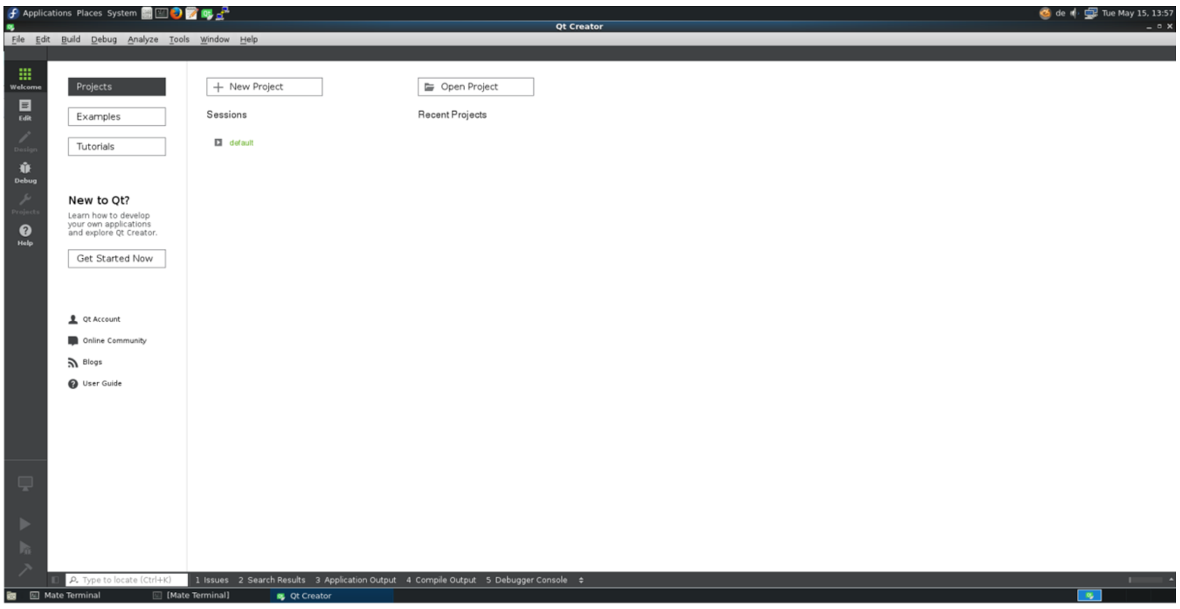


Figure 1: QT Creator

Note:

If Yocto toolchain is used, qtcreator must be started from the same terminal where SDK environment are set by command:
`source <toolchain directory>/environment-setup-aarch64-poky-linux`

See **3 Setup Application settings** on how to build and install the QT5 SDK

Alternatively, create a shell script to start both commands sequentially.

E.g.: qt5-yocto.sh

```
#!/usr/bin/sh
source <toolchain directory>/environment-setupaarch64-poky-linux
qtcreator &
```

1.2 Tested Boards and Releases

Board:	Revision:	Release:
PicoCoreMX8MM	1.20	Buildroot (fsimx8mm-B2021.06)
PicoCoreMX8MM	1.20	Yocto (fsimx8mm-Y2021.04)
ArmStoneA9	1.10	Yocto (fsimx6-Y2020.03)
ArmStoneA9	1.10	Buildroot (fsimx6-B2021.10.1)
PicoCoreMX8MPr2	1.10	Yocto (fsimx8mp-Y2023.09)

2 Remote Connection

First of all the SSH connection on the remote system will be setup.

2.1 Setup SSH connection on SBC/SOM

You have to setup your F&S Board. Therefore you can have a look into F&S *LinuxOnFS-Boards_eng.pdf*. After that boot your F&S Board.

To work with SSH the board should have a valid date. This is necessary to create certificates for SSH. To setup a date you can use the following command:

```
date "2020-05-04 10:13"
```

Afterwards we have to enable the network interface. You can also set the network on command in UBoot to enable network interface at each boot. For further information please take a look in *FSiMX6_FirstSteps_eng.pdf*.

Dynamically:

```
udhcpc -i eth0
```

Static (e.g. ip address is 10.0.0.84):

```
ifconfig eth0 10.0.0.84 up
```

The Root-Filesystem is read-only mounted, but we have to modify something in the filesystem so we need it read-writeable.

```
mount -o remount,rw /
```

Open *sshd*_config file

```
vi /etc/ssh/sshd_config
```

and edit the following lines:

```
…
PermitRootLogin yes
…
```

Optional you can also allow to login without password, but we recommend you to not do this because it's a security risk. If you want to do it anyway add the following line to *sshd_config* file.

```
…
PermitEmptyPasswords yes
…
```

Remote Connection

Make sure to use the internal sftp if no external binary is installed.

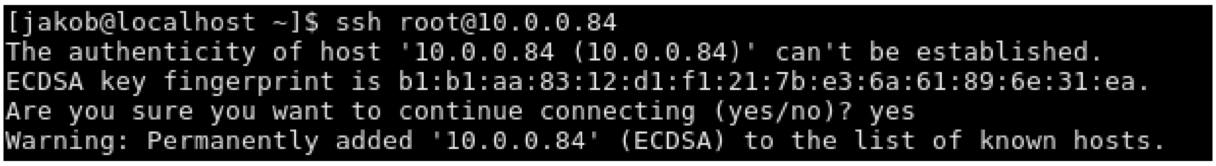
```
…
Subsystem sftp internal-sftp
…
```

After that you have to start the SSH server (This step is not required when using Yocto).

```
/etc/init.d/S50sshd start
```

Now the SSH server is running on our SBC/SOM. Let’s test it, therefore we are going back to our VM. To connect via ssh we open a Terminal and send the following command:

```
ssh root@10.0.0.84
```



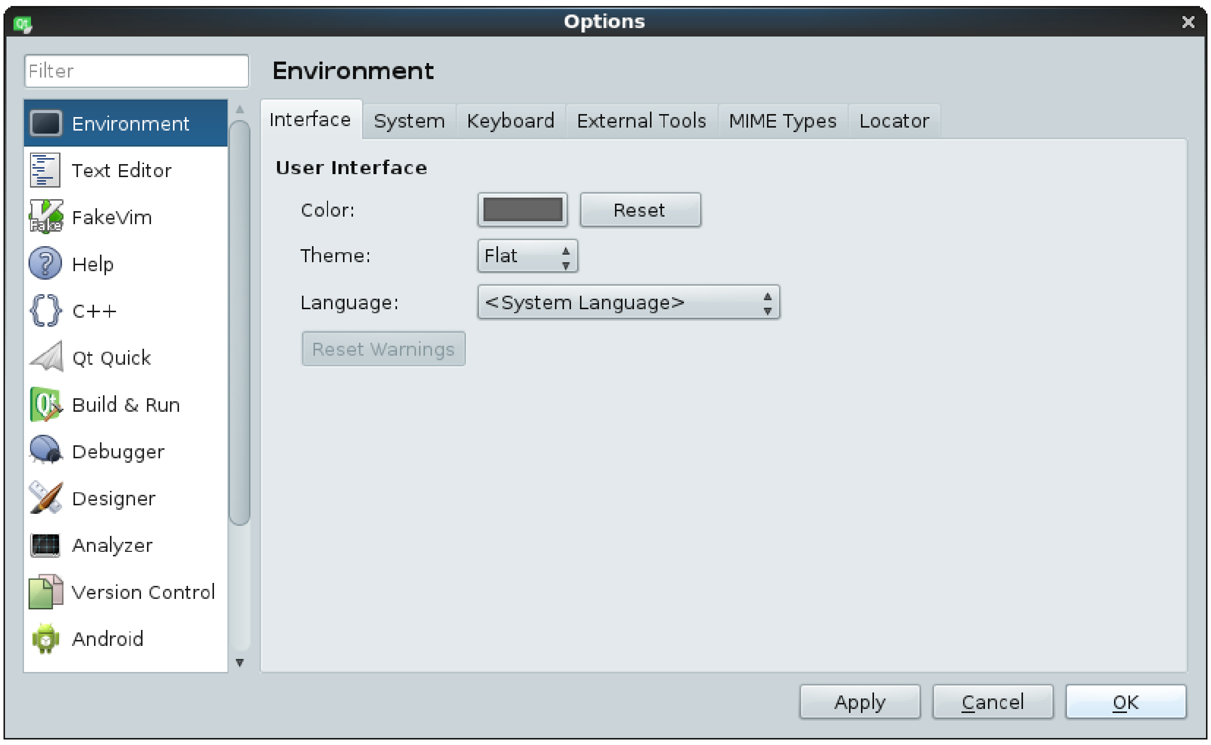
```
[jakob@localhost ~]$ ssh root@10.0.0.84
The authenticity of host '10.0.0.84 (10.0.0.84)' can't be established.
ECDSA key fingerprint is b1:b1:aa:83:12:d1:f1:21:7b:e3:6a:61:89:6e:31:ea.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.0.0.84' (ECDSA) to the list of known hosts.
```

Figure 2: SSH connection terminal

Now the SSH connection is successfully established and tested.

2.2 Setup SSH connection in Qt Creator

Open *Tools*. In the toolbar and select *Options…*.



Environment

User Interface

Color:

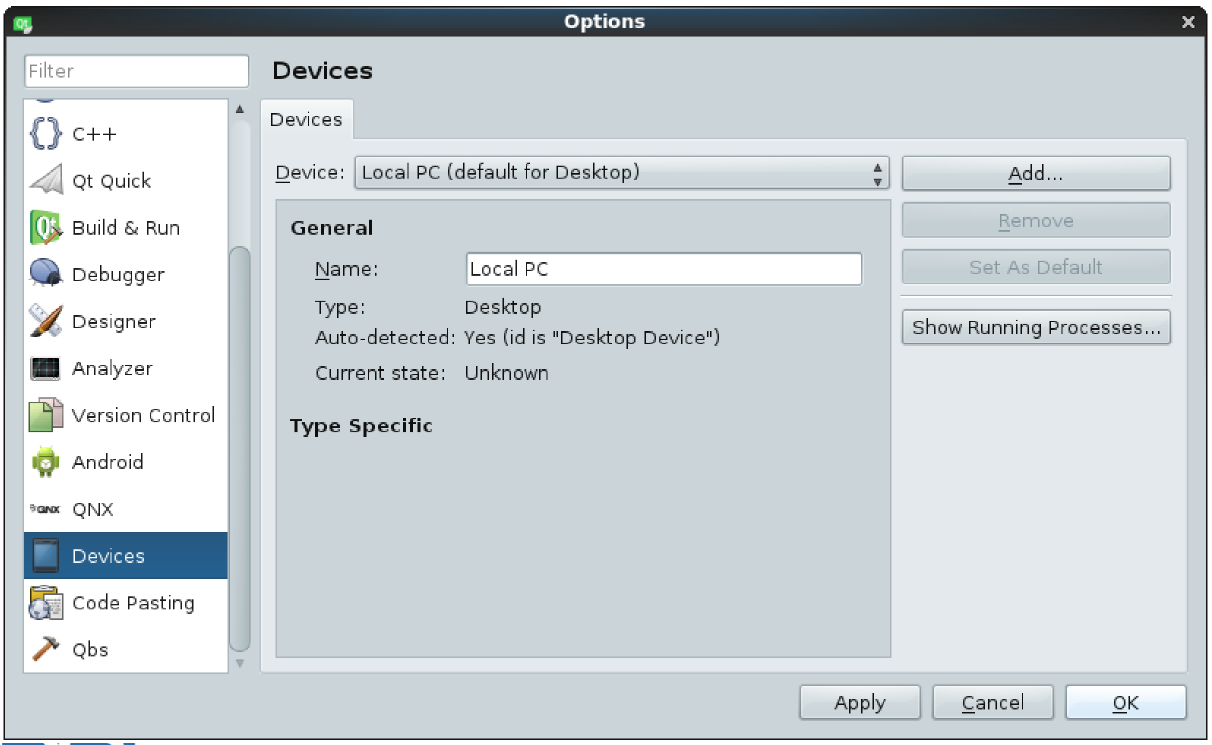
Theme: Flat

Language: <System Language>

Figure 3: Options

Figure 4: Options - Devices

Choose *Devices* on the left side.



Devices

Device: Local PC (default for Desktop)

General

Name: Local PC

Type: Desktop

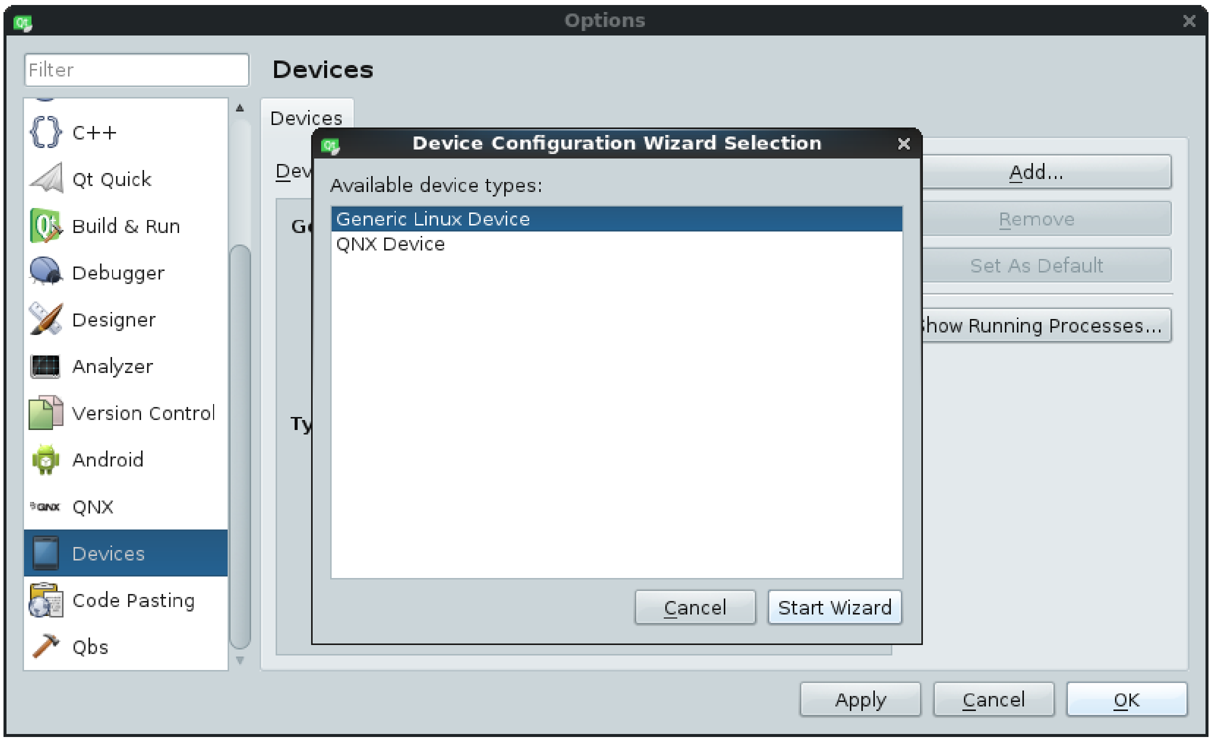
Auto-detected: Yes (id is "Desktop Device")

Current state: Unknown

Type Specific

Remote Connection

Now add a new device. Press *Add…*, choose *Generic Linux Device* and click *Start Wizard*.

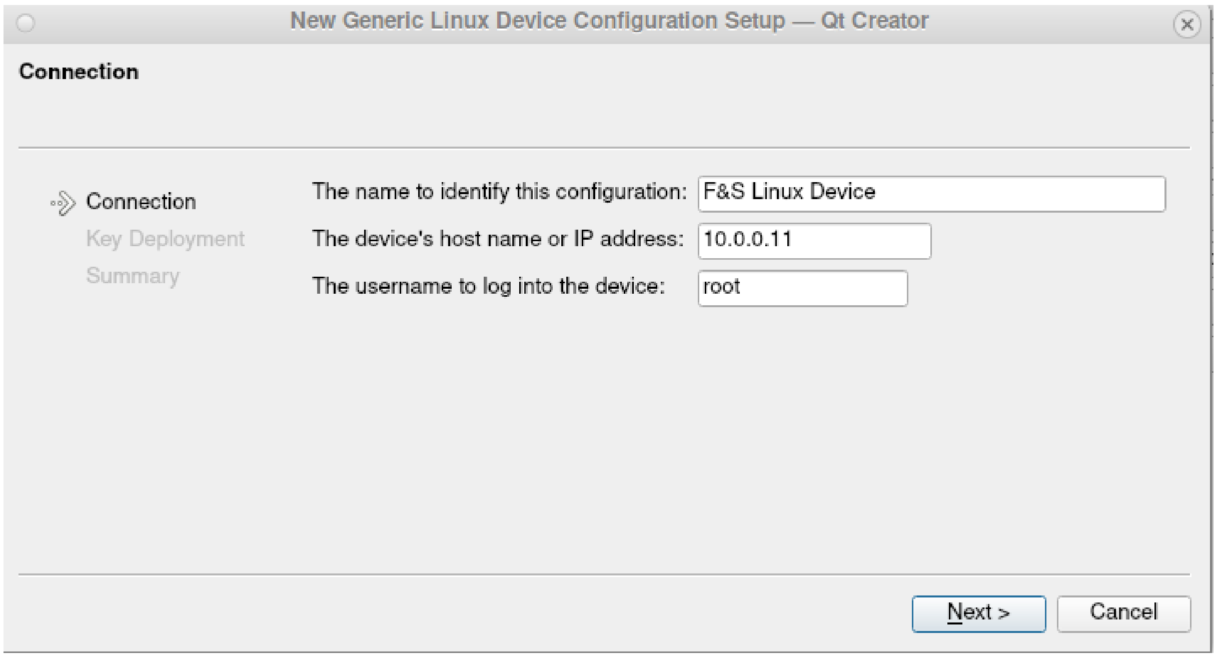


Available device types:

Generic Linux Device
QNX Device

Figure 5: Add device configuration

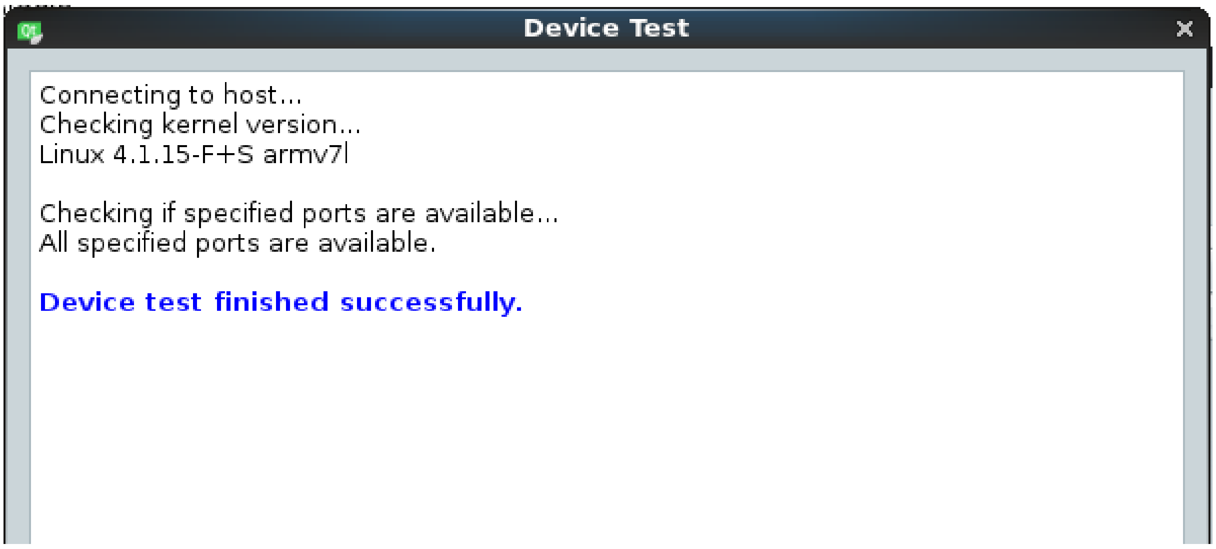
Now set a name for the configuration, enter the IP address and the login data.



The name to identify this configuration: F&S Linux Device
The device's host name or IP address: 10.0.0.11
The username to log into the device: root

Figure 6: Setup device connection

Afterwards press *Next* > *Deploy Public Key* > *Next* and *Finish*. Then the ssh connection will be automatically tested.



```
Connecting to host...
Checking kernel version...
Linux 4.1.15-F+S armv7l

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

Device test finished successfully.
```

Figure 7: Device test

3 Setup Application settings

Before you can setup the application settings be sure that you have built your Root-Filesystem (**Buildroot/Yocto**) with the correct configuration (fsimx6/ul/sx_qt5_defconfig) once. This is necessary because we need the compiled GDB debugger and other packages from this directory. If you are using Yocto you will need to create a populated SDK as well. Building the toolchain will take at least 12 GB of RAM.

Note:

If the release version which you are using is less then fsimx6-V3.1, fsimx6sx-V2.1 or fsimx6ul-V2.1, then you have to add the following packages to your root filesystem and rebuild it, e.g. through "*make menuconfig*":

- *ctest*, located in Target packages/Development Tools/ctest
- *Python support*, located in Toolchain/Build Cross gdb for the host/Python support

If you have already built your root filesystem, then you have to run the following command in your terminal:

```
…buildroot-2019.05.3-fsimx6-B2020.04]$ make host-gdb-reconfigure
…buildroot-2019.05.3-fsimx6-B2020.04]$ make
```

Populate Yocto SDK

To build and install the Yocto-SDK head over to your build directory and run the following command:

```
…<yocto-build-directory>]$ source source_env
…<yocto-build-directory>]$ bitbake -c populate_sdk <yocto-image>
```

Next you need to head to …<Build-Dir>/tmp/deploy/sdk and run the .sh script that now exist.

After that you can find your toolchain in your desired directory.

After that you have to install the new root filesystem on your remote target system.

Now select *Tools* and *Options…* in your QT Creator. Select Kits on the left side bar.

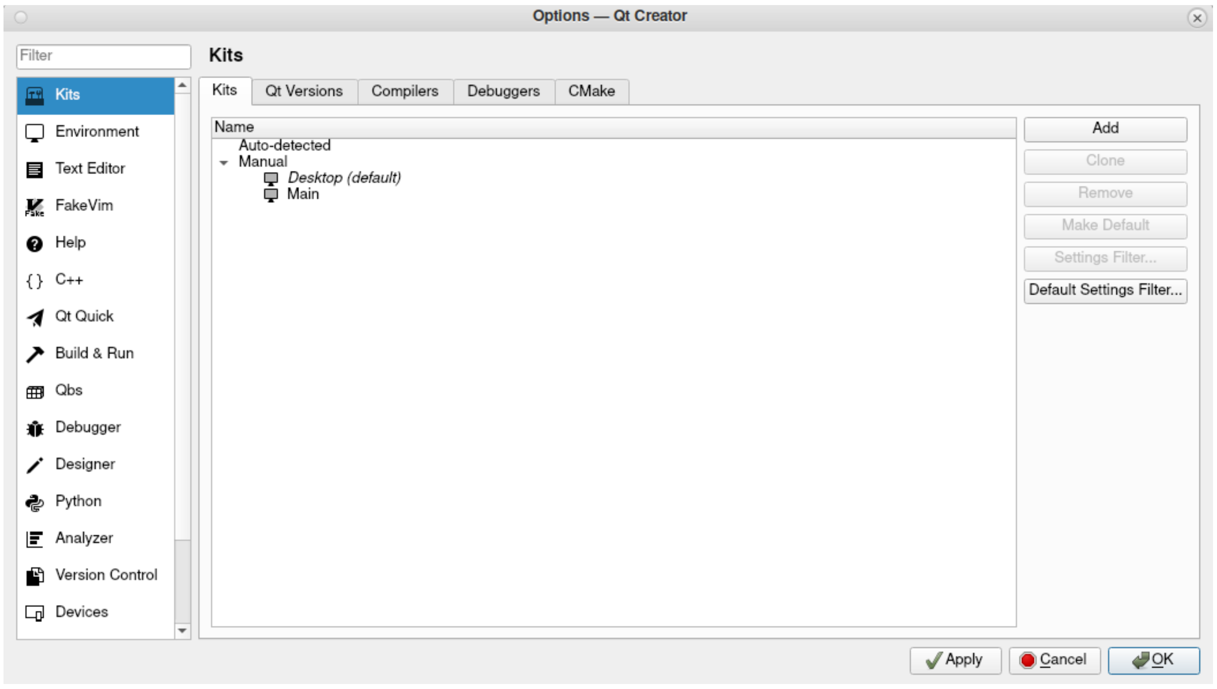
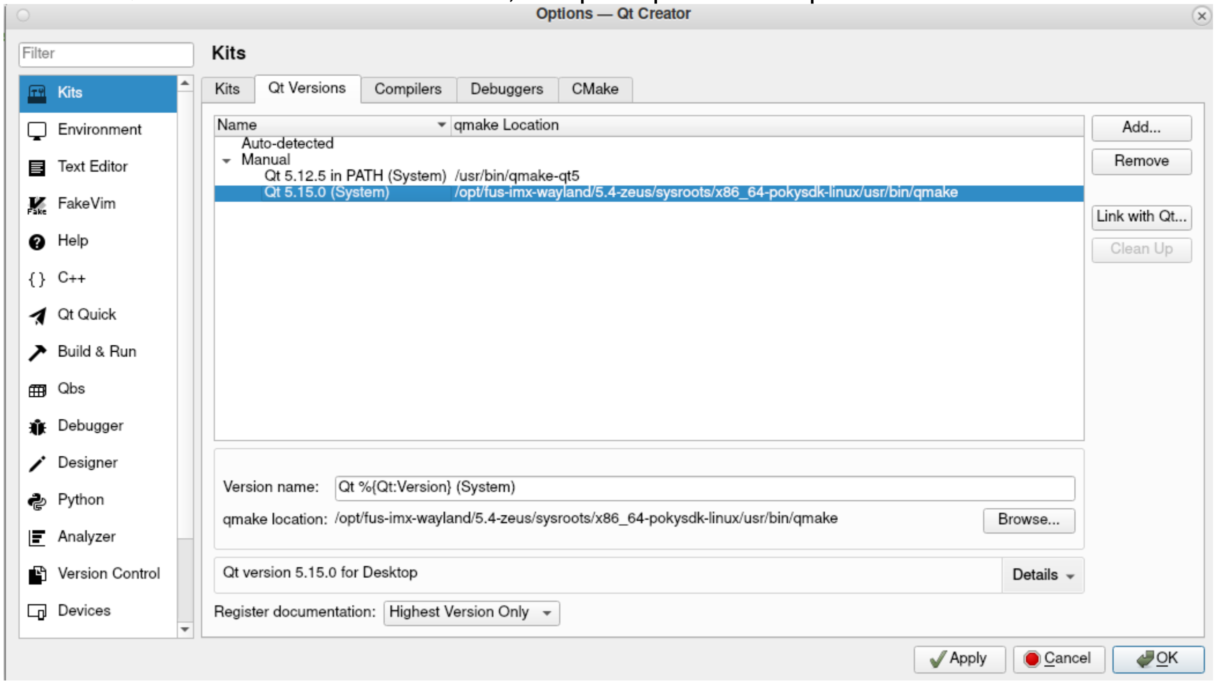


Figure 8: Options - Kits

Setup Application settings

Choose *QT Versions* and select *Add…*, setup the path to the qmake location.



Kits

Name	qmake Location
Auto-detected	
Manual	
Qt 5.12.5 in PATH (System)	/usr/bin/qmake-qt5
Qt 5.15.0 (System)	/opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/qmake

Version name: Qt %{Qt:Version} (System)

qmake location: /opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/qmake

Qt version 5.15.0 for Desktop

Register documentation: Highest Version Only

Figure 9: Kits - Qt Versions

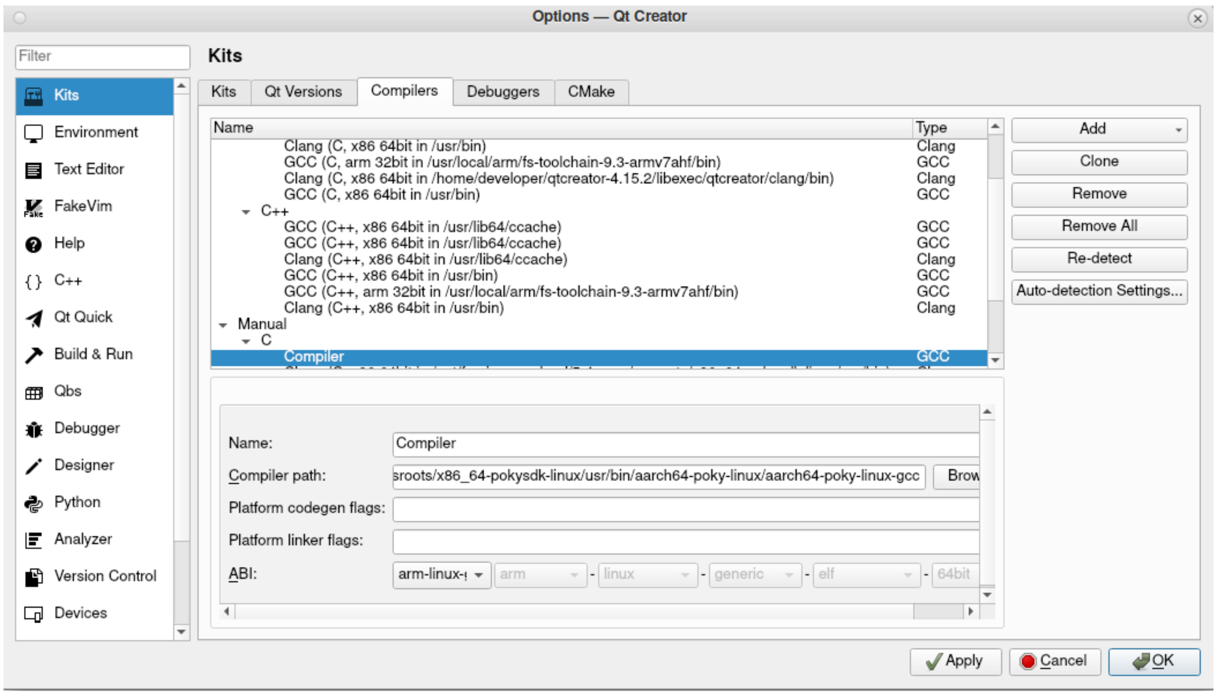
Examples:

QT5:

```
(Buildroot)
/home/developer/fsimx6-B2020.04/build/buildroot-2019.05.3-fsimx6-
B2020.04/output/host/usr/bin/qmake


(Yocto/i.MX8)
 /opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-
linux/usr/bin/qmake


(Yocto/i.MX6)
/opt/poky/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/qt5/qmake
```

QT6:

```
(Yocto/i.MX8)
/opt/fus-imx-wayland/5.15-kirkstone/sysroots/x86_64-pokysdk-
linux/usr/bin/qmake6
```



Name	Type
Clang (C, x86 64bit in /usr/bin)	Clang
GCC (C, arm 32bit in /usr/local/arm/fs-toolchain-9.3-armv7ahf/bin)	GCC
Clang (C, x86 64bit in /home/developer/qtcreator-4.15.2/libexec/qtcreator/clang/bin)	Clang
GCC (C, x86 64bit in /usr/bin)	GCC
C++	
GCC (C++, x86 64bit in /usr/lib64/ccache)	GCC
GCC (C++, x86 64bit in /usr/lib64/ccache)	GCC
Clang (C++, x86 64bit in /usr/lib64/ccache)	Clang
GCC (C++, x86 64bit in /usr/bin)	GCC
GCC (C++, arm 32bit in /usr/local/arm/fs-toolchain-9.3-armv7ahf/bin)	GCC
Clang (C++, x86 64bit in /usr/bin)	Clang
Manual	
C	
Compiler	GCC

Name:	Compiler
Compiler path:	sroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-gcc
Platform codegen flags:	
Platform linker flags:	
ABI:	arm-linux-(- arm - linux - generic - elf - 64bit

Figure 10:Kits - Compilers

Now select *Compilers* and Add *GCC then C++*. Setup the *Name* and the *Compiler path*. After that *Apply* the settings.

Examples:

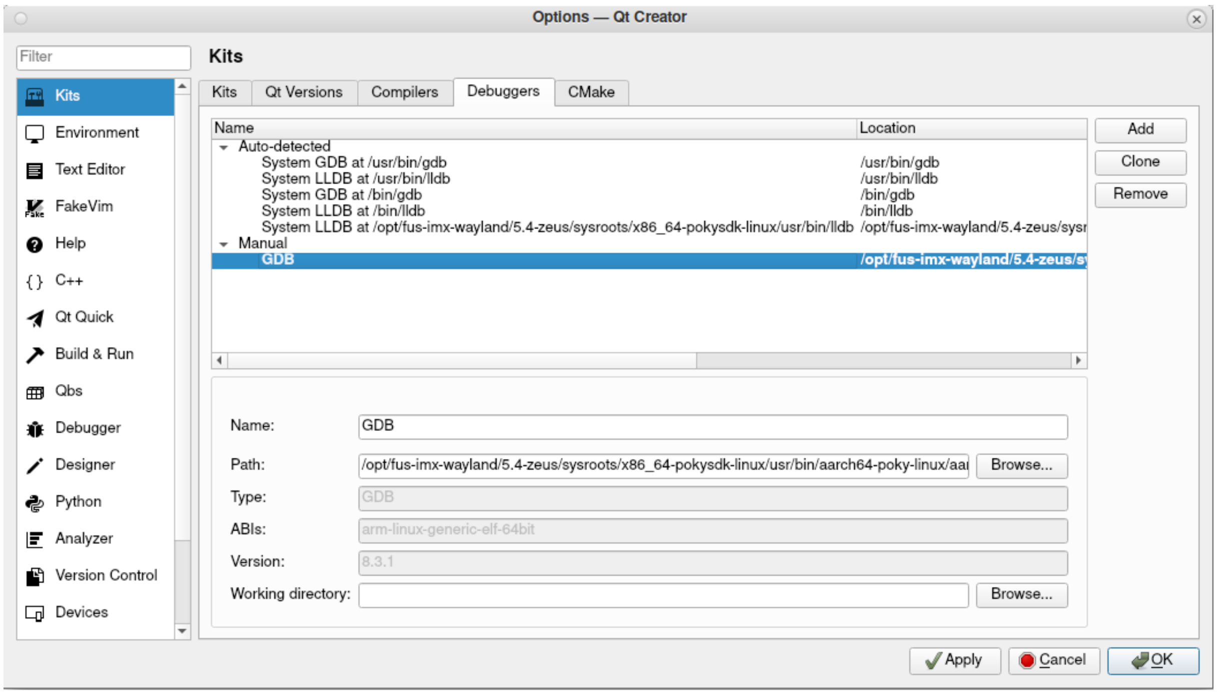
QT5:

```
(Buildroot)
/home/developer/fsimx6-B2020.04/build/buildroot-2019.05.3-fsimx6-
B2020.04/output/host/usr/bin/arm-linux-g++


(Yocto/i.MX8)
/opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-
linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-g++


(Yocto/i.MX6)
/opt/poky/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-
linux/i586-poky-linux-g++
```

QT6:

```
(Yocto/i.MX8)
/opt/fus-imx-wayland/5.15-kirkstone/sysroots/x86_64-pokysdk-
linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-g++
```

Setup Application settings



Name	Location
Auto-detected	
System GDB at /usr/bin/gdb	/usr/bin/gdb
System LLDB at /usr/bin/lldb	/usr/bin/lldb
System GDB at /bin/gdb	/bin/gdb
System LLDB at /bin/lldb	/bin/lldb
System LLDB at /opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/lldb	/opt/fus-imx-wayland/5.4-zeus/sysr
Manual	
GDB	/opt/fus-imx-wayland/5.4-zeus/s

Name: GDB
Path: /opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aar
Type: GDB
ABIs: arm-linux-generic-elf-64bit
Version: 8.3.1
Working directory:

Figure 11: Kits - Debuggers

Afterwards select *Debuggers* and *Add*. Setup *Name* and *Path* and *apply* the settings.

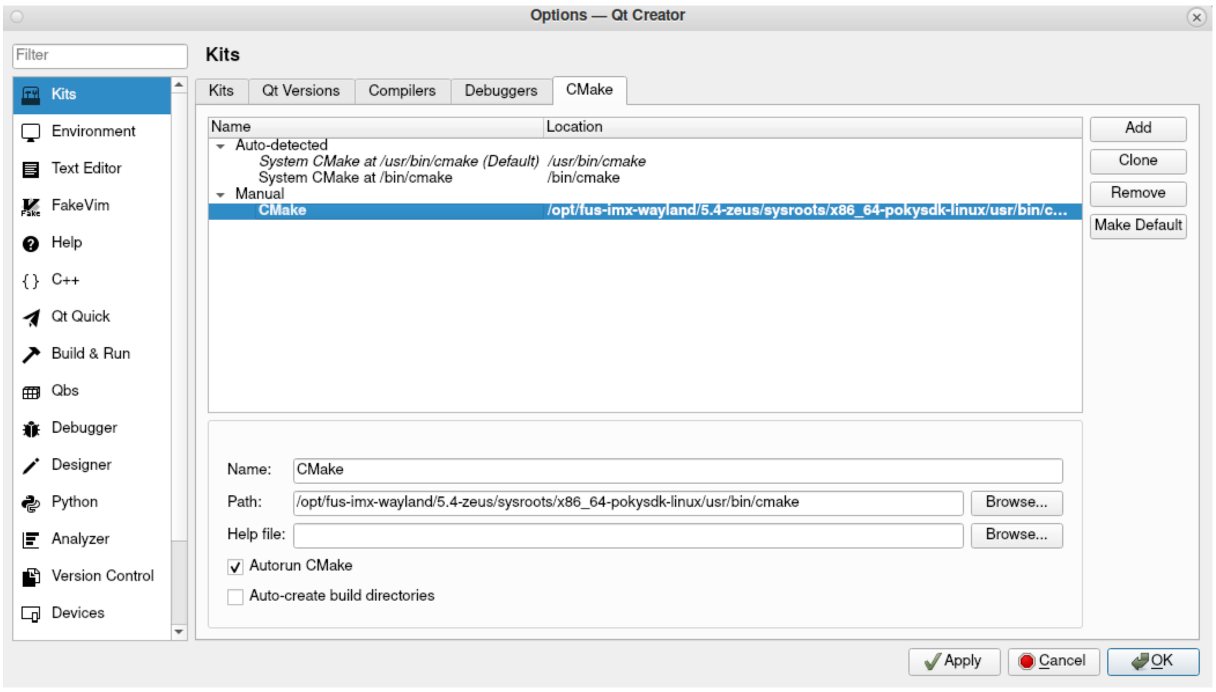
Examples:

QT5:

```
(Buildroot)
32-bit CPUs:
/home/developer/fsimx6-B2020.04/build/buildroot-2019.05.3-fsimx6-B2020.04/output/host/usr/bin/arm-linux-gdb
64-bit CPUs:
/home/developer/fsimx8mm-B2020.08/build/buildroot-2019.05.3-fsimx8mm-B2020.08/output/host//bin/aarch64-linux-gdb


(Yocto)
32-bit CPUs:
/opt/poky/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/i586-poky-linux/i586-poky-linux-gdb
64-bit CPUs:
/opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-gdb
```

QT6:

```
(Yocto)
/opt/fus-imx-wayland/5.15-kirkstone/sysroots/x86_64-pokysdk-
linux/usr/bin/aarch64-poky-linux/aarch64-poky-linux-gdb
```



Name	Location
Auto-detected	
System CMake at /usr/bin/cmake (Default)	*/usr/bin/cmake*
System CMake at /bin/cmake	/bin/cmake
Manual	
CMake	/opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/c...

Name: CMake
Path: /opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-linux/usr/bin/cmake
Help file:
[✓] Autorun CMake
[ ] Auto-create build directories

Figure 12: Kits – Cmake

After that select *CMake* and *Add*. Setup *Name* and Path and *Apply* the settings.

Examples:

QT5:

```
(Buildroot)
/home/developer/fsimx6-B2020.04/build/buildroot-2019.05.3-fsimx6-
B2020.04/output/host/usr/bin/cmake


(Yocto/.MX8)
/opt/fus-imx-wayland/5.4-zeus/sysroots/x86_64-pokysdk-
linux/usr/bin/cmake


(Yocto/i.MX6)
```

Setup Application settings

```
/opt/poky/2.4.2/sysroots/x86_64-pokysdk-linux/usr/bin/cmake
```

QT6:

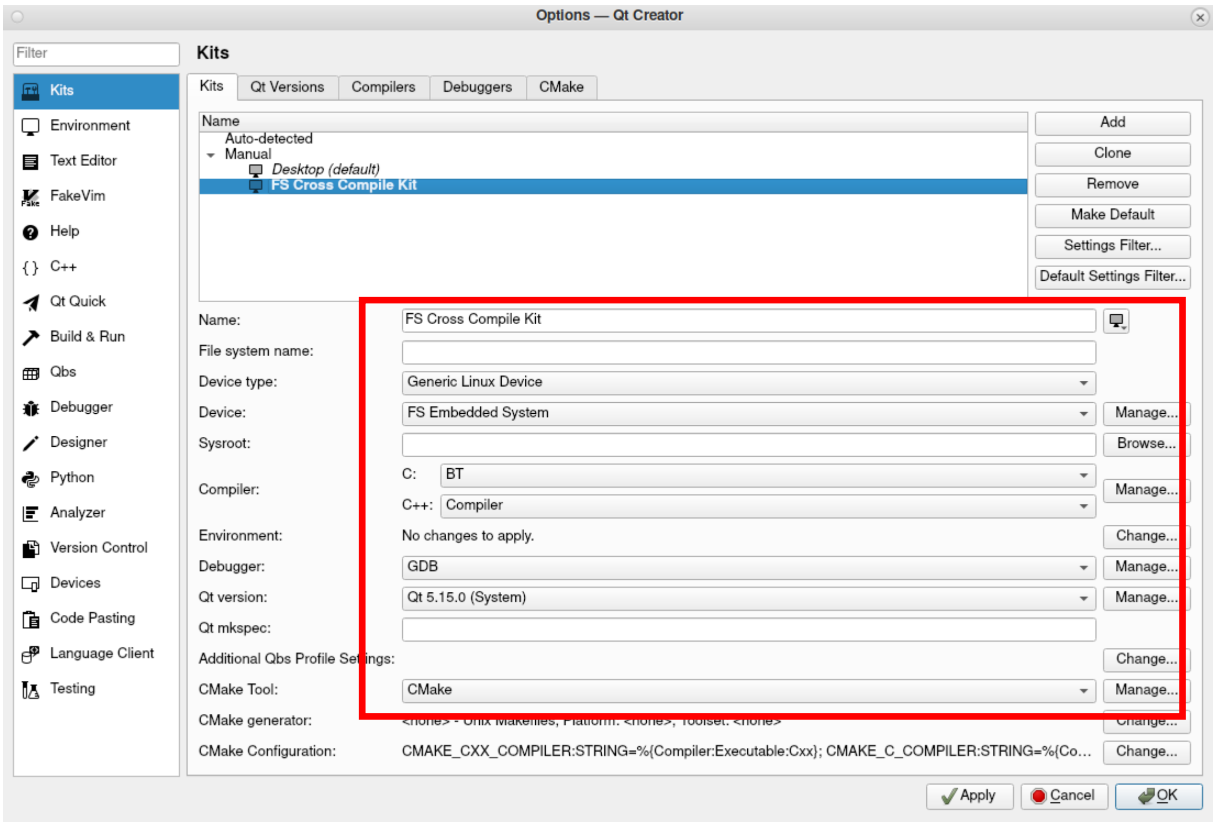
```
(Yocto/i.MX8)
/opt/fus-imx-wayland/5.15-kirkstone/sysroots/x86_64-pokysdk-
linux/usr/bin/cmake
```

Now everything is setup and the Kit can be created. Select *Kits* and *Add*. Setup the Kit (refer to figure 13).

Note:

If you are unable to find the file *cmake* in buildroot directory output/host/usr/bin, then you have to add the following package to your buildroot configuration and rebuild it, e.g. through "*make menuconfig*":

- *host cmake*, located in Host utilities/host cmake



Name: FS Cross Compile Kit
File system name:
Device type: Generic Linux Device
Device: FS Embedded System
Sysroot:
Compiler: C: BT
C++: Compiler
Environment: No changes to apply.
Debugger: GDB
Qt version: Qt 5.15.0 (System)
Qt mkspec:
Additional Qbs Profile Settings:
CMake Tool: CMake
CMake generator: <none> - Unix Makefiles, Platform: <none>, Toolset: <none>
CMake Configuration: CMAKE_CXX_COMPILER:STRING=%{Compiler:Executable:Cxx}; CMAKE_C_COMPILER:STRING=%{Co...

Figure 13: Build & Run - Kits

Select *Apply* and *OK* to finish the setup.

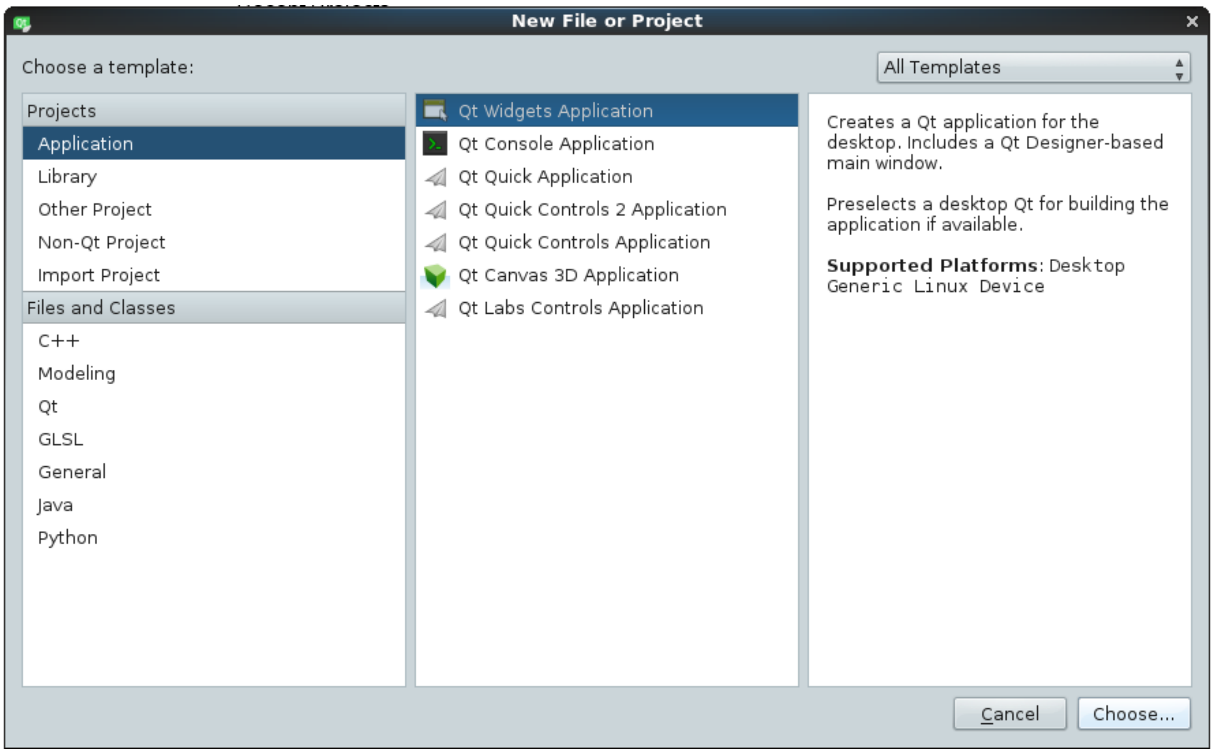
4 Create an Application

QT5

After everything is setup, you can create a new project. From *File* select *New File or Project…*. Select *Application* and *QT Widgets Application* and click *Choose…*.

QT6

After everything is setup, you can create a new project. From File select *New Project…* *Select Application (Qt)* and *Qt Widgets Application* and click *Choose…*



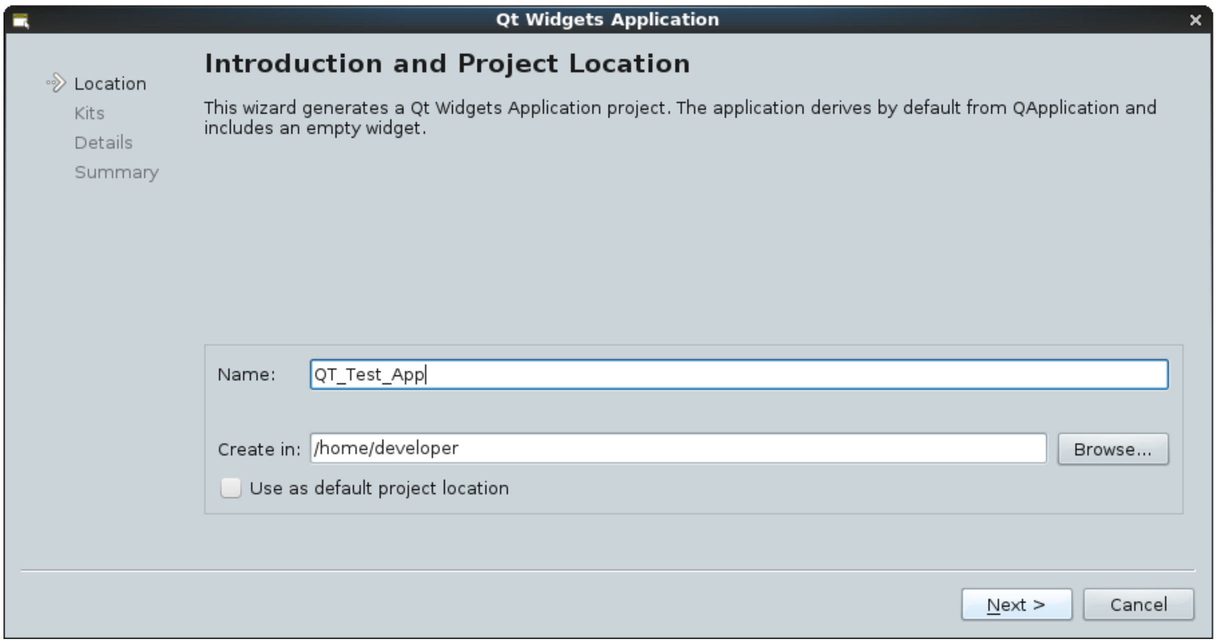
Creates a Qt application for the desktop. Includes a Qt Designer-based main window.

Preselects a desktop Qt for building the application if available.

Supported Platforms: Desktop Generic Linux Device

Figure 14: New File or Project

Choose a name of the project and the location of the project. Press *Next >*.



Introduction and Project Location

This wizard generates a Qt Widgets Application project. The application derives by default from QApplication and includes an empty widget.

Name: QT_Test_App

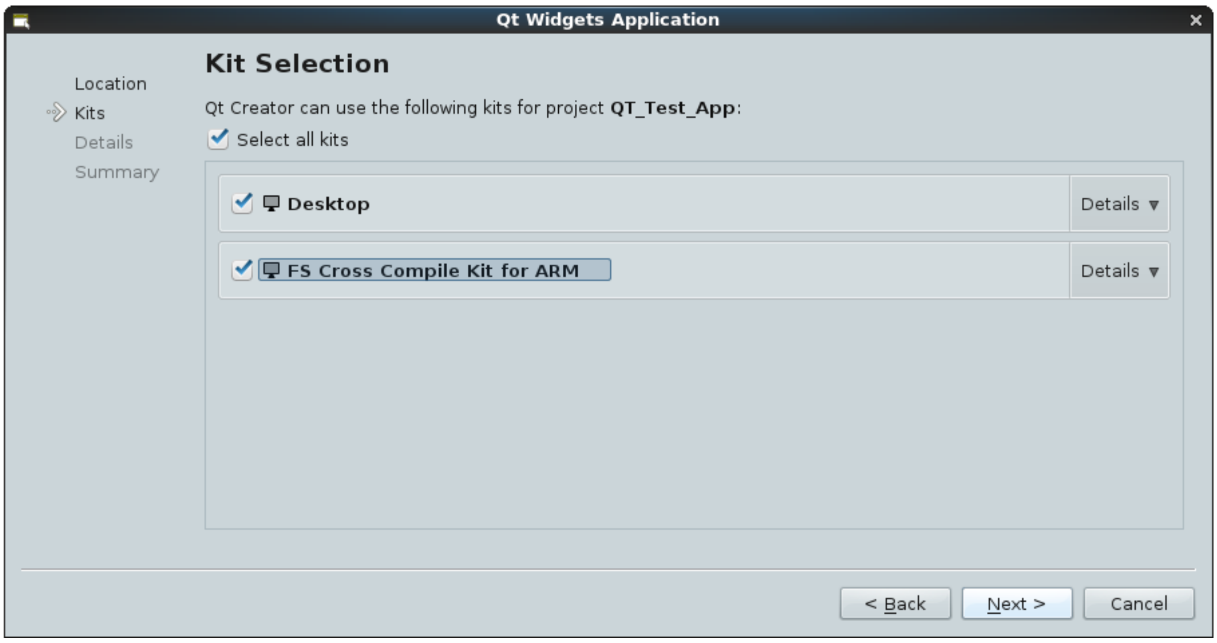
Create in: /home/developer

Use as default project location

Figure 15: Introduction and Project Location

QT5:

Select the Kits which you want to use and press *Next >*.



Kit Selection

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

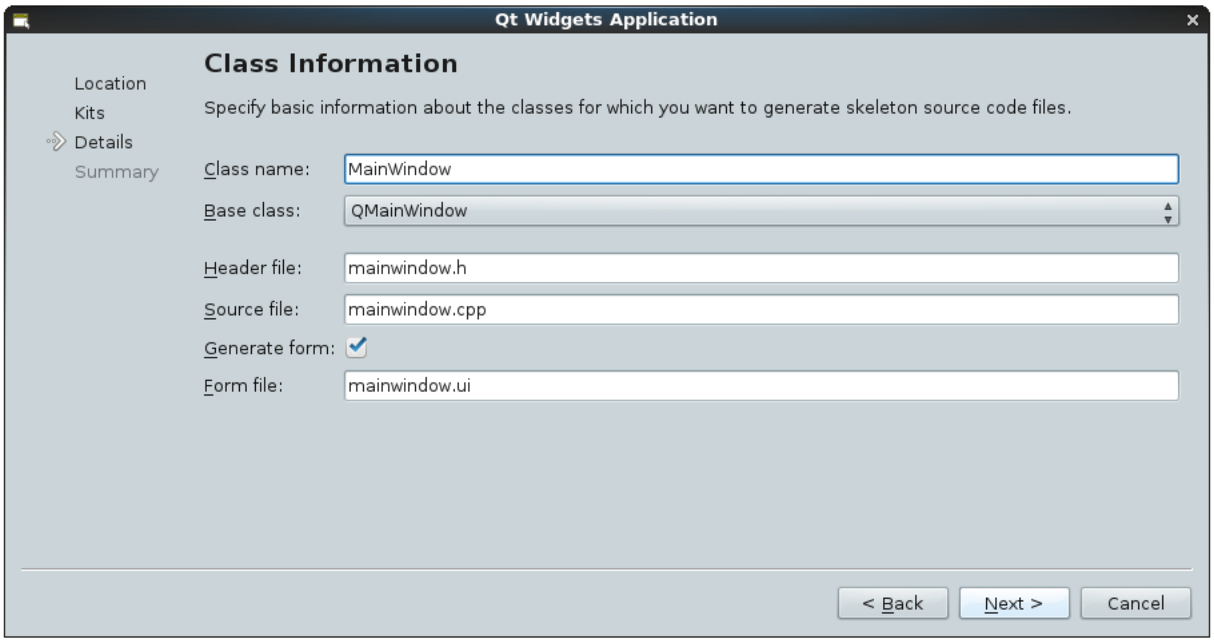
✓ Select all kits

✓ **Desktop**

✓ **FS Cross Compile Kit for ARM**

Figure 16: Kit Selection

Choose class information's and press *Next >*.



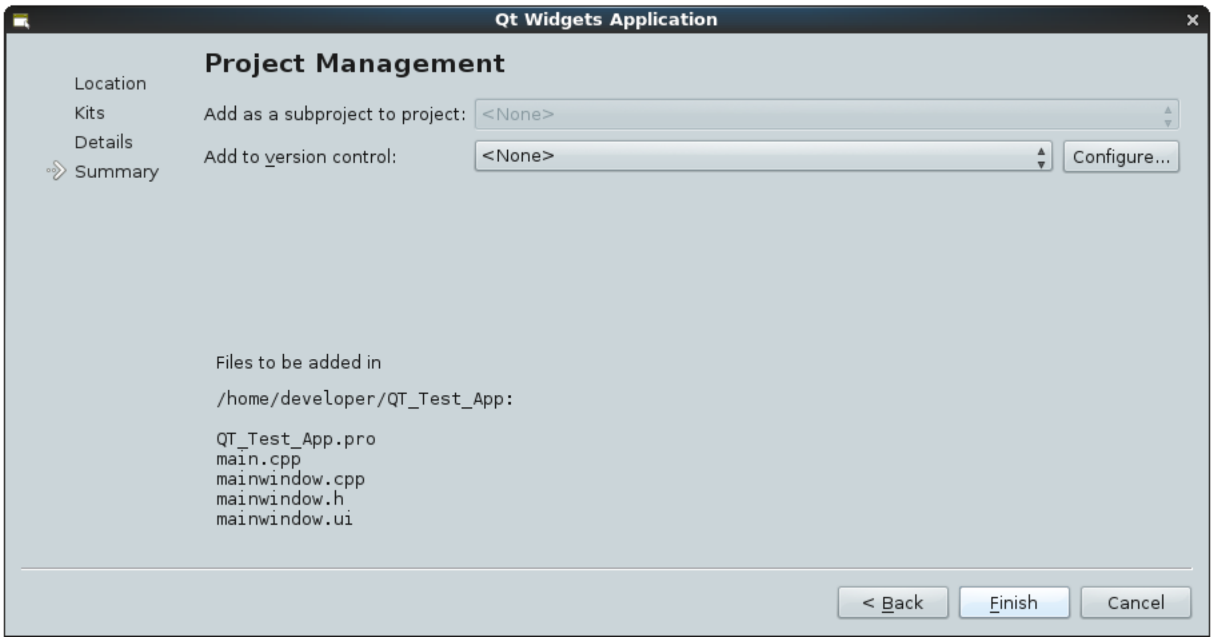
Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name:	MainWindow
Base class:	QMainWindow
Header file:	mainwindow.h
Source file:	mainwindow.cpp
Generate form:	✓
Form file:	mainwindow.ui

Figure 17: Class Information

Select Project Management and click *Finish*.



Add as a subproject to project: <None>

Add to version control: <None>

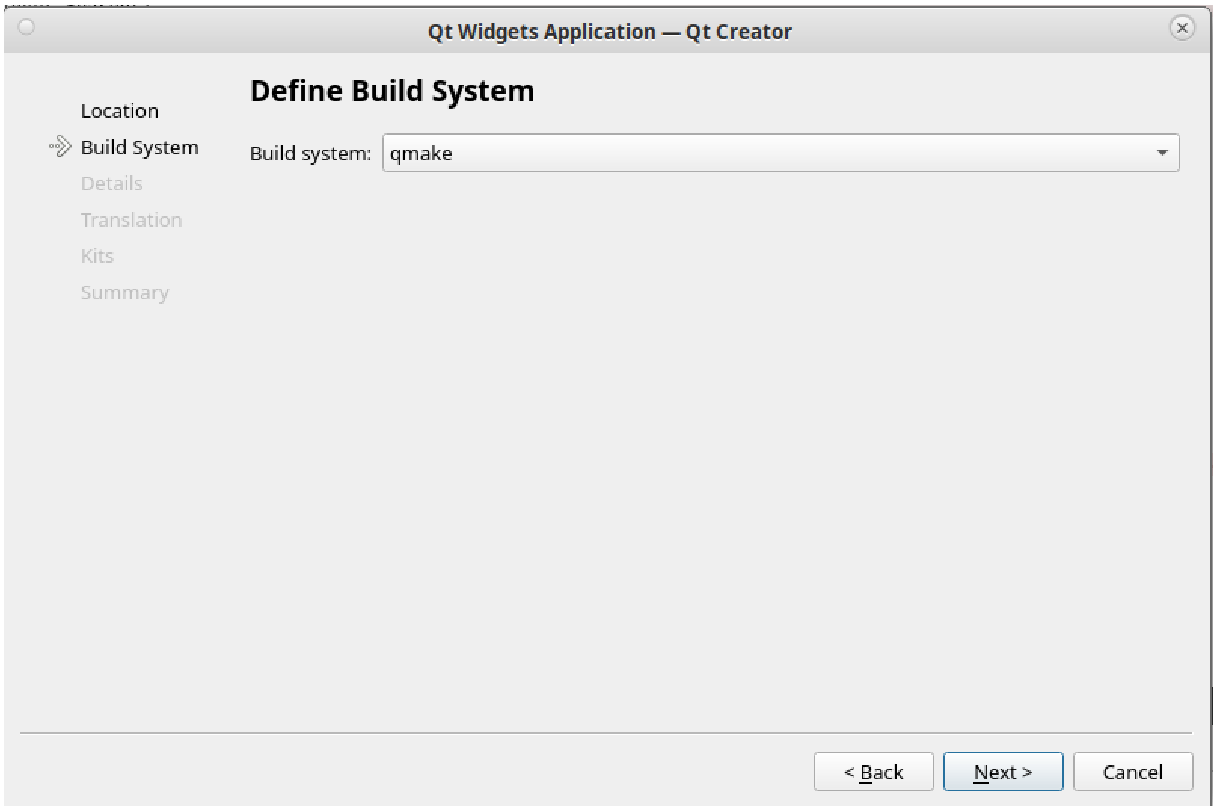
Files to be added in

```
/home/developer/QT_Test_App:

QT_Test_App.pro
main.cpp
mainwindow.cpp
mainwindow.h
mainwindow.ui
```

Figure 18: Project Management

QT6:

Choose *qmake* and press *Next >*.

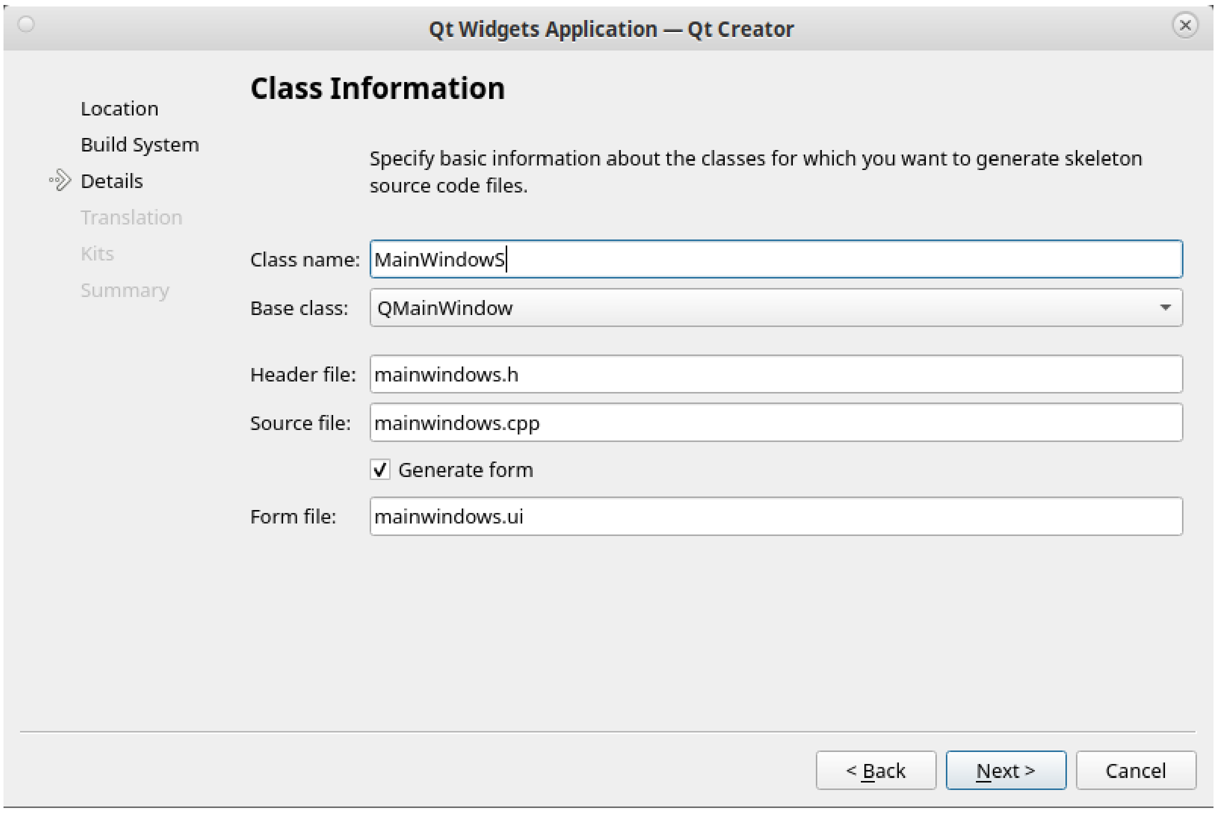

Qt Widgets Application — Qt Creator

Define Build System

Build system: qmake

Figure 19: Define Build System

Choose class information’s and press *Next >*.



Qt Widgets Application — Qt Creator

Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name: MainWindowS

Base class: QMainWindow

Header file: mainwindows.h

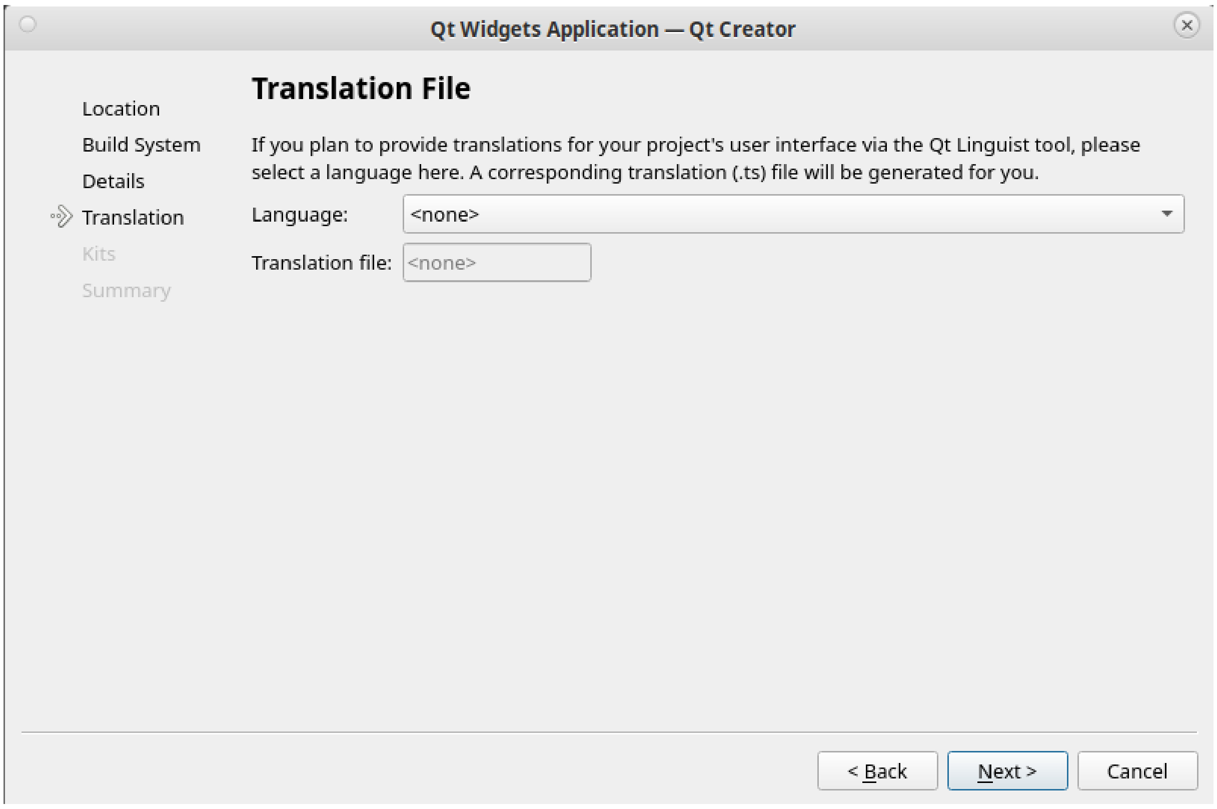
Source file: mainwindows.cpp

☑ Generate form

Form file: mainwindows.ui

Figure 20: Class Information QT6

Choose your Translation File or leave it blank. Press *Next >.*



Qt Widgets Application — Qt Creator

Translation File

If you plan to provide translations for your project's user interface via the Qt Linguist tool, please select a language here. A corresponding translation (.ts) file will be generated for you.

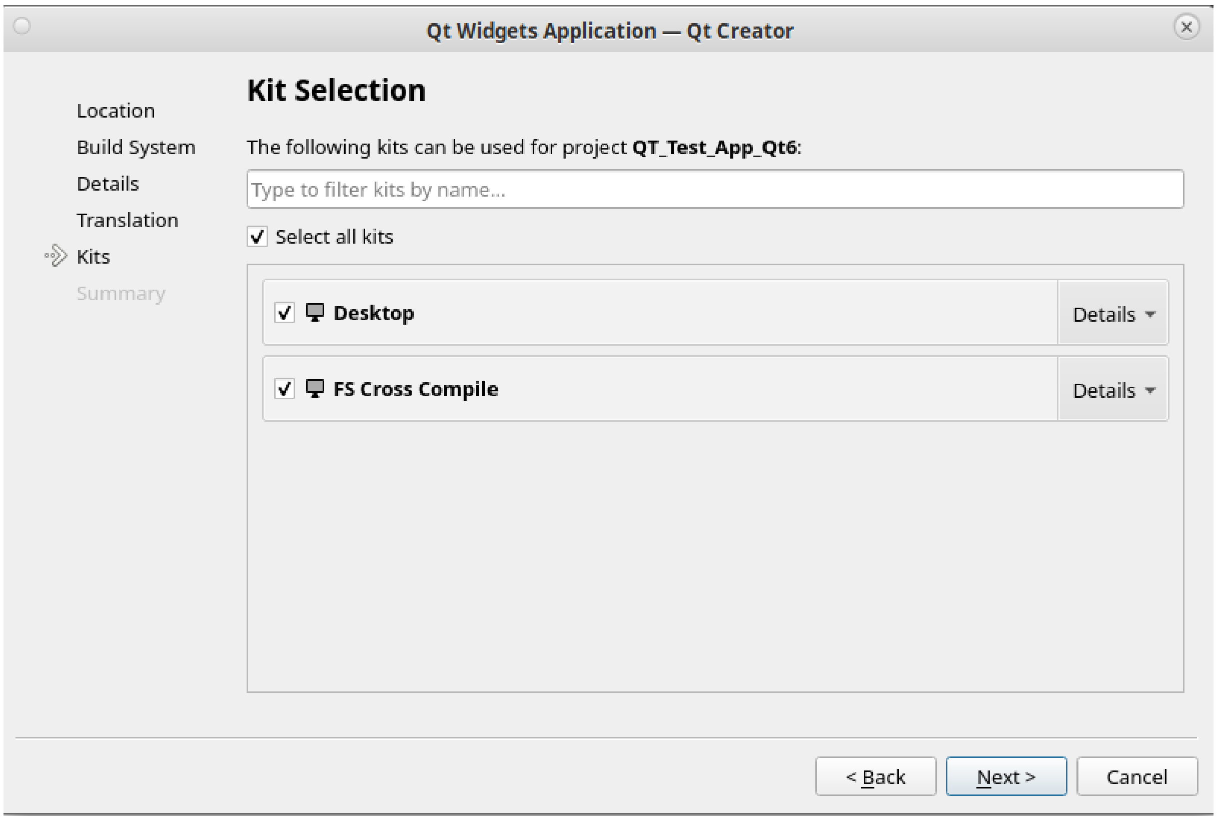
Language: <none>

Translation file: <none>

Figure 21: Translation File

Create an Application

Select the Kits which you want to use and press *Next >*.


Qt Widgets Application — Qt Creator

Location
Build System
Details
Translation
Kits
Summary

Kit Selection

The following kits can be used for project **QT_Test_App_Qt6**:

Type to filter kits by name...

☑ Select all kits

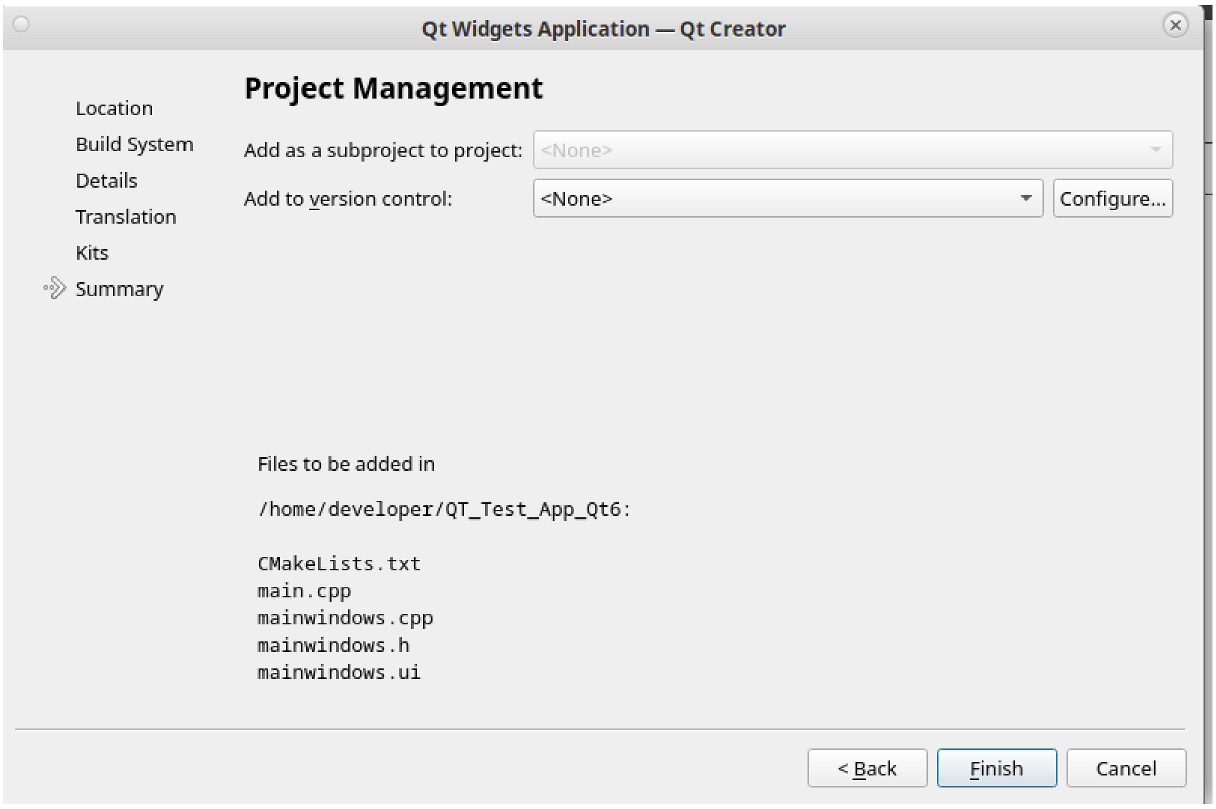
☑ **Desktop** Details

☑ **FS Cross Compile** Details

< Back | Next > | Cancel

Figure 22: Kit Selection QT6

Click *Finish.*



Project Management

Add as a subproject to project: <None>

Add to version control: <None>

Files to be added in

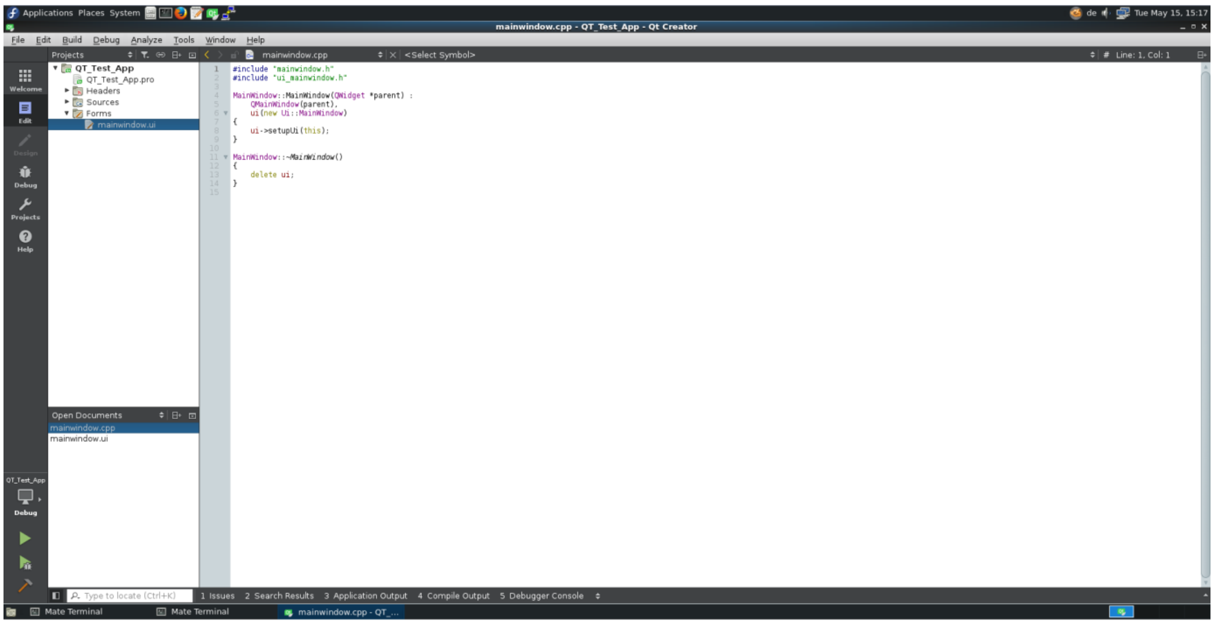
```
/home/developer/QT_Test_App_Qt6:

CMakeLists.txt
main.cpp
mainwindows.cpp
mainwindows.h
mainwindows.ui
```

Figure 23: Project Management QT6

Now the project is successfully created.

4.1 Create hello world app

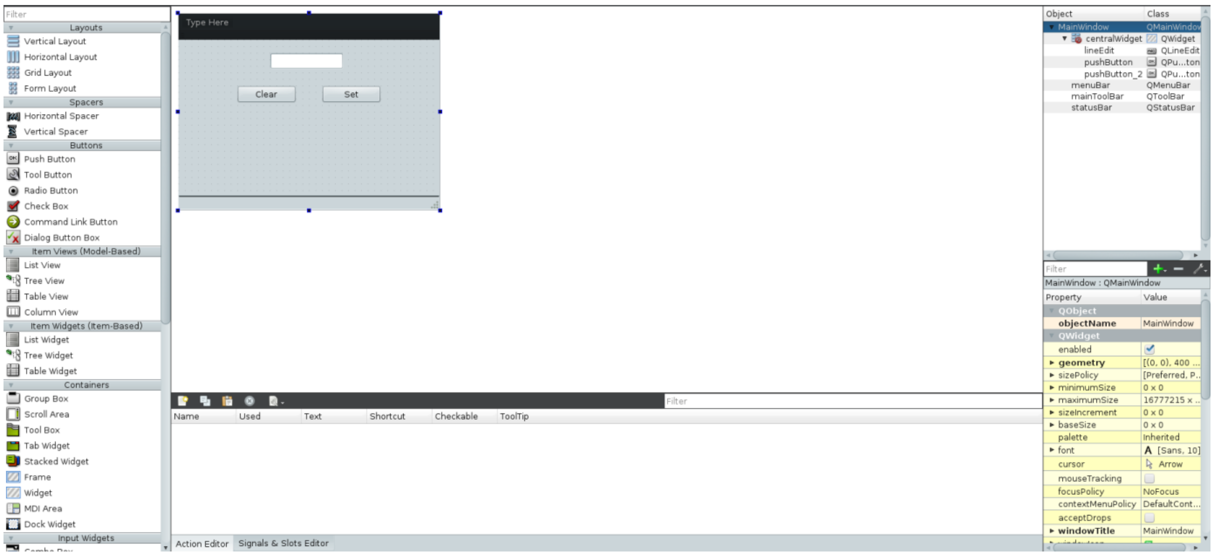
Open the folder Forms on the left side in project view. Double click the mainwindow.ui.



```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Figure 24: Project file - mainwindow.cpp

Now move 2 Push Buttons to the Main Window and move one Line Edit (refer to figure 20). After that change the name of the buttons with a double click on the button.



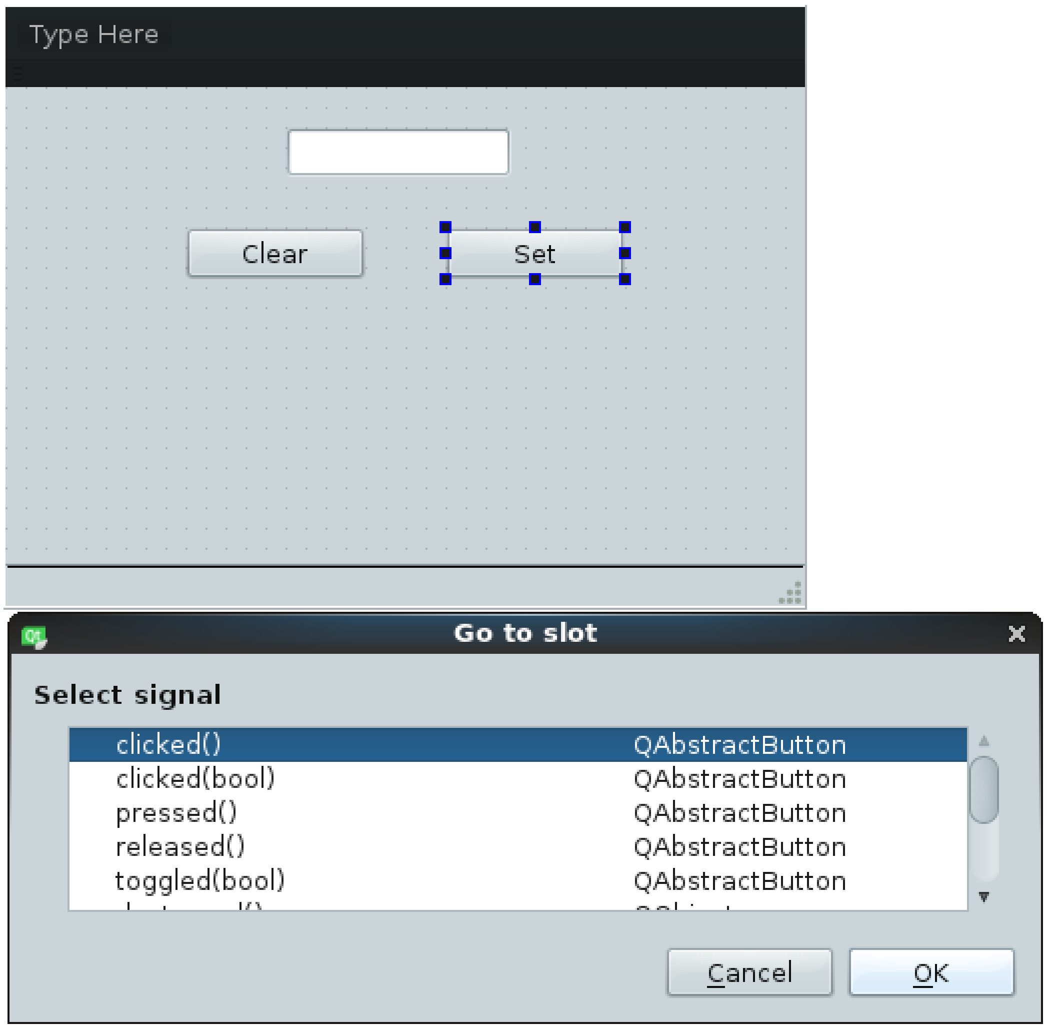
Object	Class
MainWindow	QMainWindow
centralWidget	QWidget
lineEdit	QLineEdit
pushButton	QPu...ton
pushButton_2	QPu...ton
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

MainWindow : QMainWindow

Property	Value
QObject	
objectName	MainWindow
QWidget	
enabled	✓
geometry	[(0, 0), 400 ...
sizePolicy	[Preferred, P...
minimumSize	0 x 0
maximumSize	16777215 x ...
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	[Sans, 10]
cursor	Arrow
mouseTracking	☐
focusPolicy	NoFocus
contextMenuPolicy	DefaultCont...
acceptDrops	☐
windowTitle	MainWindow

Figure 25: Project file - mainwindow.ui

After that right click the “Set” button and select *Go to slot…*. Then select the signal *clicked()* and press *OK*.



clicked()	QAbstractButton
clicked(bool)	QAbstractButton
pressed()	QAbstractButton
released()	QAbstractButton
toggled(bool)	QAbstractButton

Figure 26: Select signal

The view changes to the cpp file. It automatically creates a function called *void MainWindow::on_pushButton_clicked()*. Now add the following line in the created function.

```
…
void MainWindows::on_pushButton_clicked()
{
    ui->lineEdit->setText(“Hello World!”);
}
```

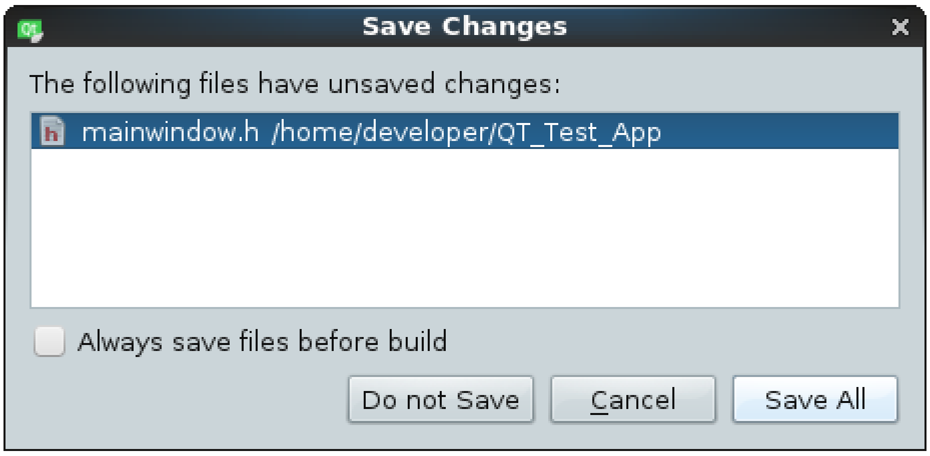
Now double click mainwindow.ui again. After that right click the “Clear” button and select *Go to slot…*. Then select the signal *clicked()* and press *OK*. After that the view changes to the cpp file again. It automatically creates a function called *void MainWindow::on_pushButton_2_clicked()*. Now add the following line in the created function.

```
…
void MainWindows::on_pushButton_2_clicked()
{
    ui->lineEdit->setText(“”);
}
```

5 Build and Debug Application

Select *Build* in the Toolbar and press *Build Project “QT_Test_App”*. Otherwise you can click the build icon in the left toolbar.

If you have unsaved files then the system ask you what to do with these unsaved changes. We press Save All, because we want to save these modifications.



Save Changes

The following files have unsaved changes:

mainwindow.h /home/developer/QT_Test_App

Always save files before build

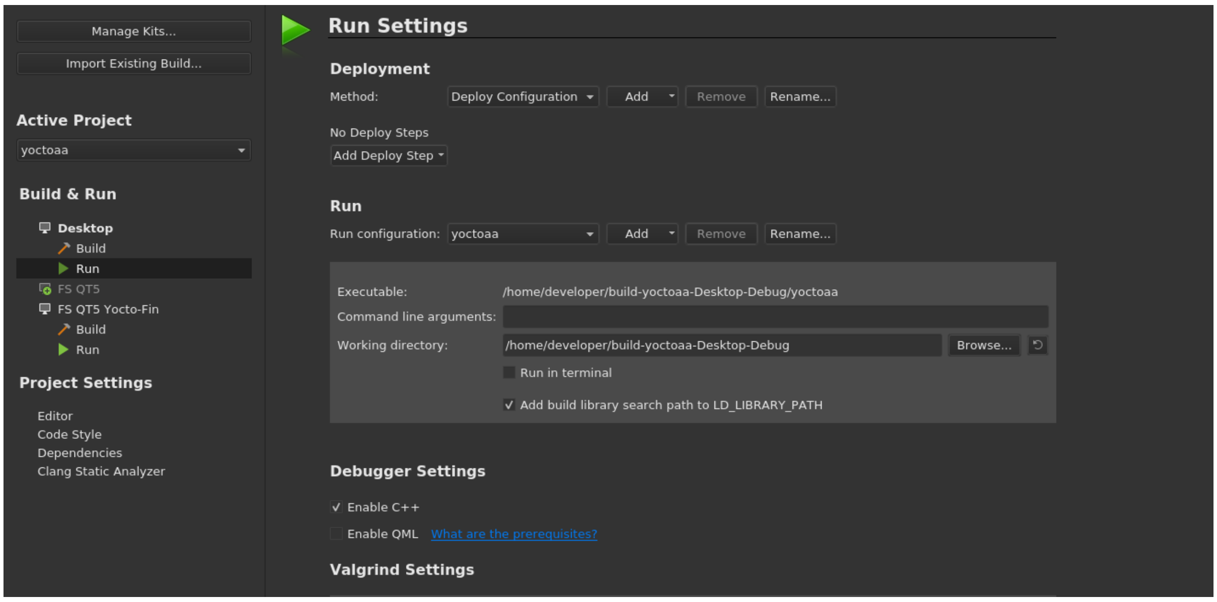
Figure 27: Save changes

To be sure everything is successfully built select at the bottom bar *4 Compile Output* and have a look at the output message.

5.1 Desktop Run

After everything is successfully built, select *Projects* on the left sidebar. After that select *Run* in the Desktop Kit.

If you don't see the kit in the tab, select *Add Kit* and add your corresponding Kit.



Run Settings

Deployment

Method: Deploy Configuration

No Deploy Steps

Run

Run configuration: yoctoaa

Executable: /home/developer/build-yoctoaa-Desktop-Debug/yoctoaa

Command line arguments:

Working directory: /home/developer/build-yoctoaa-Desktop-Debug

Run in terminal

✓ Add build library search path to LD_LIBRARY_PATH

Debugger Settings

✓ Enable C++

Enable QML What are the prerequisites?

Valgrind Settings

Active Project: yoctoaa

Figure 28: Project - Desktop Run

Then select *Build* on the toolbar and click *Run*. Otherwise you can click the Run icon on the left toolbar.

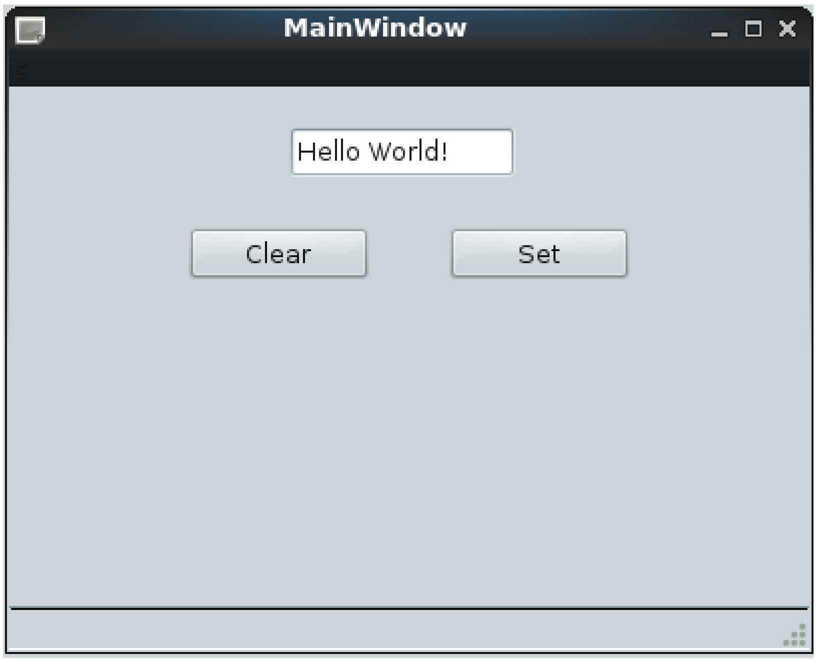


Figure 29: Application

Now the application will be executed on your development machine. Test the app by clicking the *Set* and the *Clear* buttons.

You can click the x in the right corner of the application to stop it.

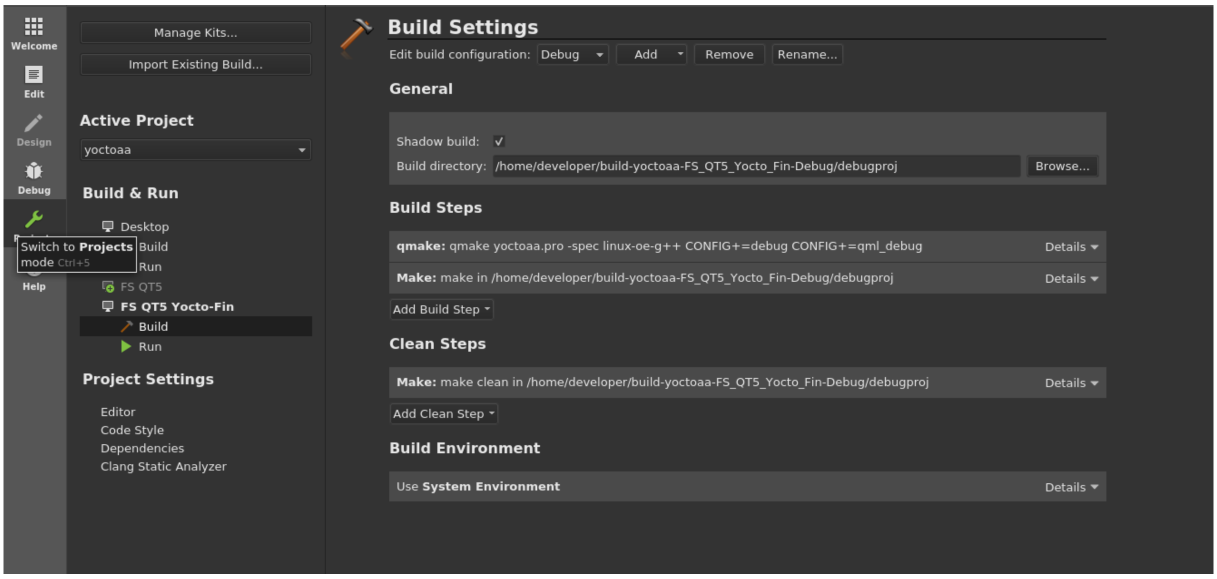
5.2 FS Cross Compile Kit for ARM Run

First of all we have to setup something in our project. Select .pro file in my case QT_Test_App.pro and add at the end following lines:

```
…
linux-* {
# Buildroot
    target.path = /root
# Yocto
    target.path = /home/root
    INSTALLS += target
}
```

Now save the changes and select *Projects* on the left toolbar.

To run the compiled example for the ARM system, select Build in the FS Cross Compile Kit for ARM. Then select *Browse…* and create a folder and set the Build directory.



Build Settings

Edit build configuration: Debug

General

Shadow build: ✓

Build directory: /home/developer/build-yoctoaa-FS_QT5_Yocto_Fin-Debug/debugproj

Build Steps

qmake: qmake yoctoaa.pro -spec linux-oe-g++ CONFIG+=debug CONFIG+=qml_debug

Make: make in /home/developer/build-yoctoaa-FS_QT5_Yocto_Fin-Debug/debugproj

Clean Steps

Make: make clean in /home/developer/build-yoctoaa-FS_QT5_Yocto_Fin-Debug/debugproj

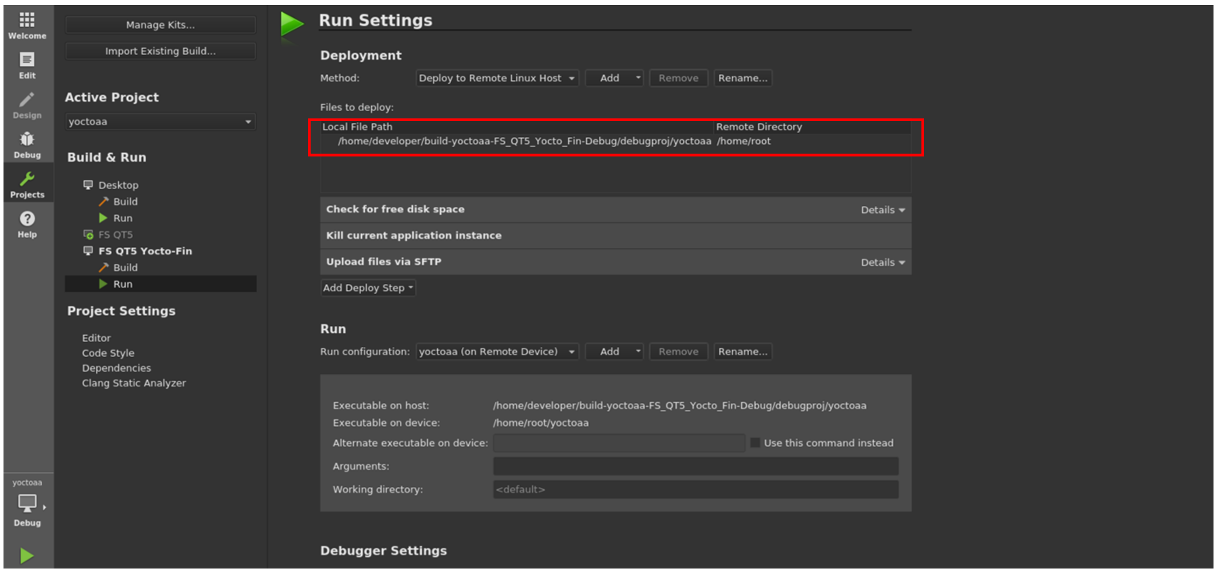
Build Environment

Use **System Environment**

Figure 30: Project - Build FS Cross Compile Kit for ARM

Then build the system by clicking *Build* in the toolbar and select *Build Project “QT_Test_App”*. Otherwise you can click the build icon in the left toolbar.

Then select *Run* in the FS Cross Compile Kit for ARM. Normally it should automatically add the files to deploy with the correct local path and the remote directory.



Run Settings

Deployment

Method: Deploy to Remote Linux Host

Files to deploy:

Local File Path	Remote Directory
/home/developer/build-yoctoaa-FS_QT5_Yocto_Fin-Debug/debugproj/yoctoaa	/home/root

Check for free disk space

Kill current application instance

Upload files via SFTP

Run

Run configuration: yoctoaa (on Remote Device)

Executable on host: /home/developer/build-yoctoaa-FS_QT5_Yocto_Fin-Debug/debugproj/yoctoaa
Executable on device: /home/root/yoctoaa
Alternate executable on device:
Arguments:
Working directory: <default>

Debugger Settings

Active Project: yoctoaa

Figure 31: Project - Run FS Cross Compile Kit for ARM

Now everything is setup and you can run your application on your embedded device. Click *Build* in the Toolbar and select *Run*. Otherwise you can click the run icon on the left toolbar.

Now the application is running on your system.

5.3 Debug build

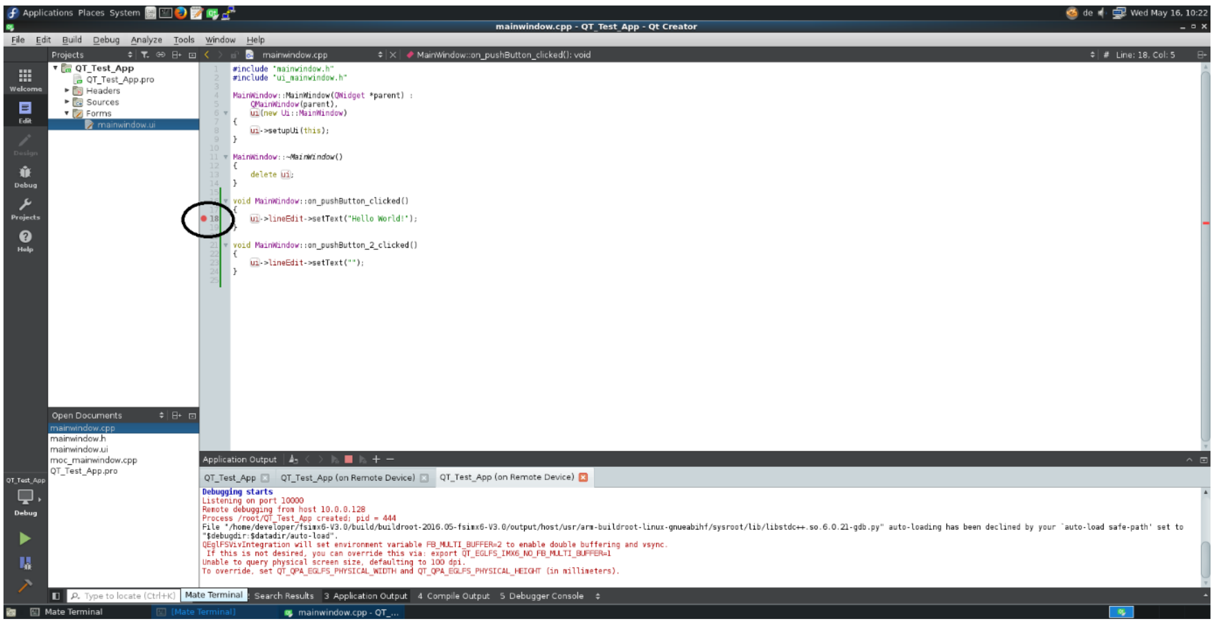
To debug the application you have to choose *Debug* in the toolbar, *Start Debugging* and *Start Debugging* again instead of Build/Run. Otherwise you can click the debug icon on the left toolbar.

Now you can debug your application on your system.

Note:

If you are using a backend like wayland / X11 you have to fetch the environment variables from the embedded device in QT-Creator. Select Projects on the left sidebar. After that select Run in the FS Cross Compile Kit for ARM. Now scroll down to Run Environment and press Fetch Device Environment. Now you should see the environment of the embedded device. It is necessary that the SSH connection works.

To setup debug points select your c file which you want to debug and left click on the line in your program, where you want to set a debug point. Click Debug in the toolbar to see the different step options.



```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_pushButton_clicked()
{
    ui->lineEdit->setText("Hello World!");
}

void MainWindow::on_pushButton_2_clicked()
{
    ui->lineEdit->setText("");
}
```

```
Debugging starts
Listening on port 10000
Remote debugging from host 10.0.0.128
Process /root/QT_Test_App created; pid = 444
File "/home/developer/fsimx6-V3.0/build/buildroot-2016.05-fsimx6-V3.0/output/host/usr/arm-buildroot-linux-gnueabihf/sysroot/lib/libstdc++.so.6.0.21-gdb.py" auto-loading has been declined by your `auto-load safe-path' set to
"$debugdir:$datadir/auto-load".
QEglFSVivIntegration will set environment variable FB_MULTI_BUFFER=2 to enable double buffering and vsync.
 If this is not desired, you can override this via: export QT_EGLFS_IMX6_NO_FB_MULTI_BUFFER=1
Unable to query physical screen size, defaulting to 100 dpi.
To override, set QT_QPA_EGLFS_PHYSICAL_WIDTH and QT_QPA_EGLFS_PHYSICAL_HEIGHT (in millimeters).
```

Figure 32: Debug Application

6 Build CMAKE Projects

This guide explains how to build CMake-based Qt projects using the Yocto populated SDK within Qt Creator.

6.1 Background

Unlike qmake, where references to dependent libraries are automatically known through the toolchain, CMake does not include these references by default. However, the Yocto SDK provides a CMake toolchain file that contains all necessary paths and settings. This file can easily be integrated into your project's CMAKE configuration.

6.2 Prerequisites

- Installed and configured Yocto SDK (populated SDK)
- Qt Creator
- A CMake -based Qt project

6.3 Step-by-Step Instructions

Carry out the following steps:

1. **Import the Qt Project:**
   Launch Qt Creator and import your existing Qt project based on CMAKE.

2. **Configure the Toolchain Settings**
   Go to the following settings in Qt Creator:
   Menu: **Projects**
   Selection: **CMake**
   Tab: **Initial Configuration**

3. **Add the Following Entries**
   Replace <yocto-version> with the appropriate version of your Yocto SDK.

```
CMAKE_FIND_ROOT_PATH = /opt/fus-imx-wayland/<yocto-
version>/sysroots/x86_64-pokysdk-linux/


CMAKE_TOOLCHAIN_FILE = /opt/fus-imx-wayland/<yocto-
version>/sysroots/x86_64-pokysdk-
linux/usr/share/cmake/OEToolchainConfig.cmake
```

7 Appendix

List of Figures

Important Notice