Developing with SailfishOS

a short introduction

Sven Putze, hardcodes.de

December 19, 2013

hardcodes.de

Contents

1 About

Hi, my name is Sven[hc01] and I am eager to develop for the Jolla smartphone. On my way some questions came up and I tried to answer them as good as I can. After a while I decided to write down what I've learned, so that I had a central place to come back to and maybe others can benefit from this small document, too. The latest and greatest version is to be found on Github[hc02]. For those who don't want to clone the git project there will be just the pdf version at `http://hardcodes.de/SailfishOS/Developing-with-SailfishOS.pdf`.

If you find typos, errors or quirks, have suggestions how to make this document better, please drop me a note. Or collaborate. `#jolla2gether`!

Don't panic if you are not comfortable with writing in LaTeX, I am happy to use your Libre/Open Office or even Word documents. The source will be converted to `MultiMarkDown` in near future[1] to lower the barrier for contribution. This way you can use your editor of choice and must not learn too many obscure instructions.

2 Download and install

You will need Virtual Box, because some components of the SailfishOS SDK come as virtual machines. Download for your development machine[vbox01].

[1] whatever *near* means.

Figure 1: Download VirtualBox from the virtualbox website.

If you already use VirtualBox, you don't need to load it again, just skip that step. Just make sure that you have the latest updates installed.

To take a dip, head over to the Sailfish Website[sailfishos01] and download the SDK for your operating system.

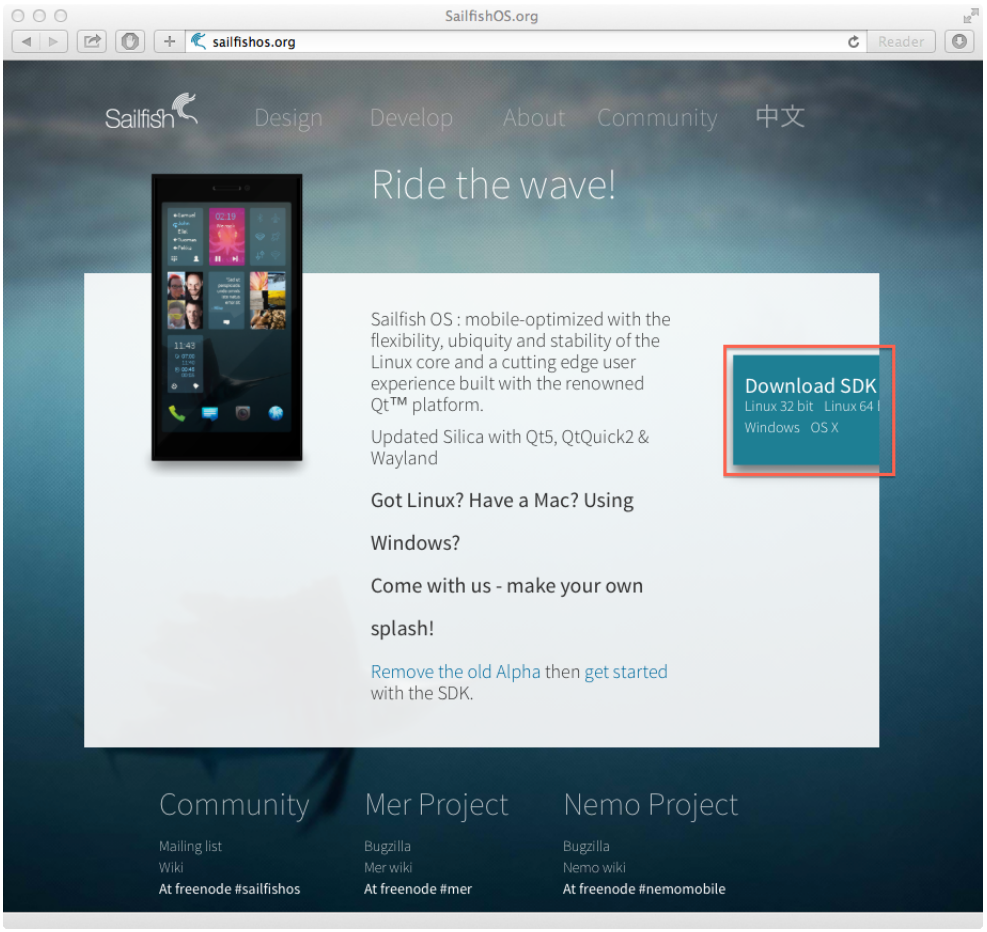


Figure 2: Download the SDK from the SailfishOS website.

2.1 OSX

Double click on the downloaded disk image for VirtualBox and run the installer application inside. Some file go into system folders and you must be or elevate to an admin account to install successfully.

Figure 3: Downloaded diskimage.

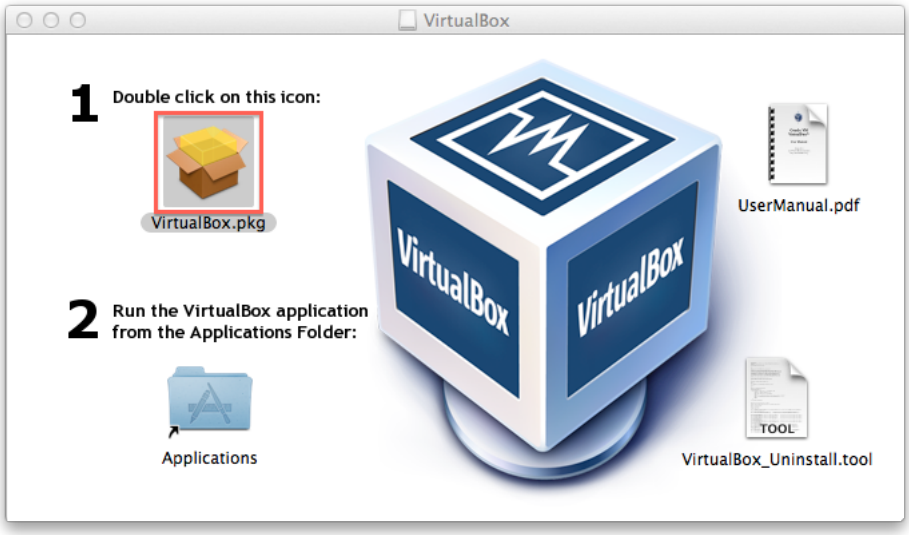


Figure 4: Install VirtualBox, step 1.

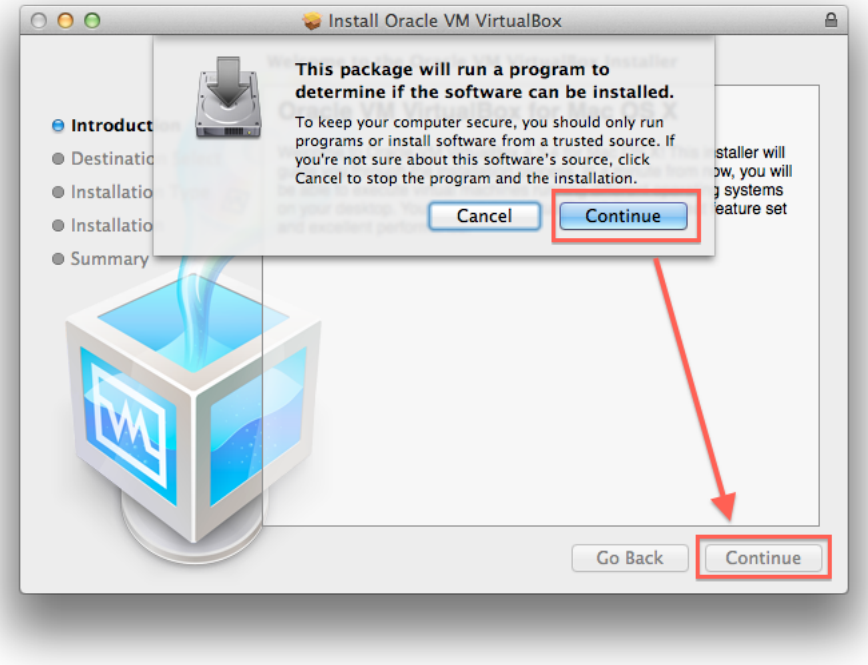


Figure 5: Install VirtualBox, step 2.

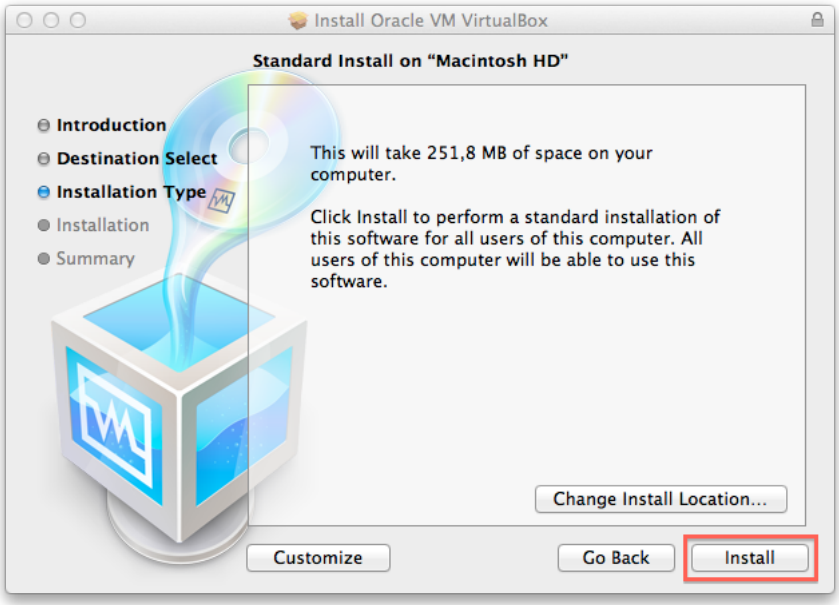


Figure 6: Install VirtualBox, step 3.

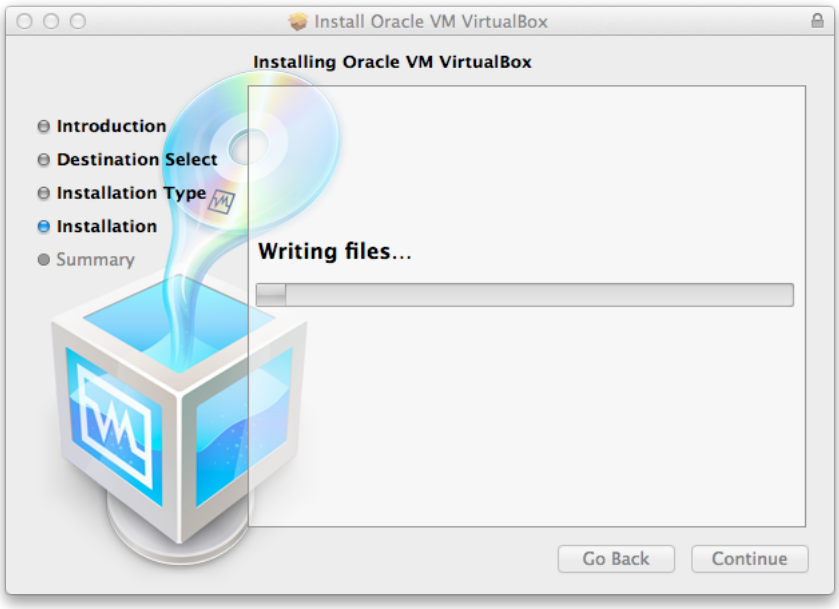


Figure 7: Install VirtualBox, step 4.

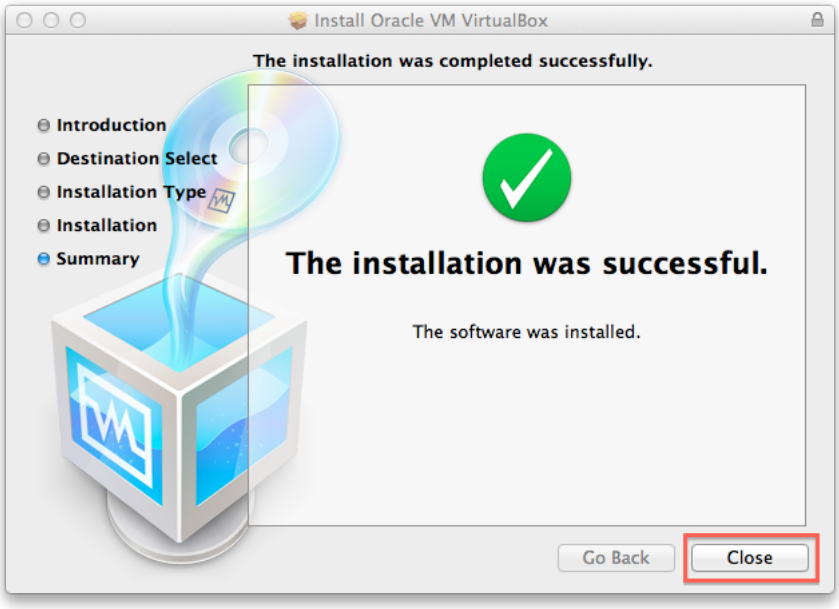


Figure 8: Install VirtualBox, step 5.

If you use VirtualBox just for the SailfishOS SDK you don't have to care about the VirtualBox application, although you can see which folders are shared inside

the preferences.

Double click on the downloaded disk image for the SailfishOS SDK and run the installer app that's inside. The installed files will end in your user directory, you don't need to be an administrator to achieve that.

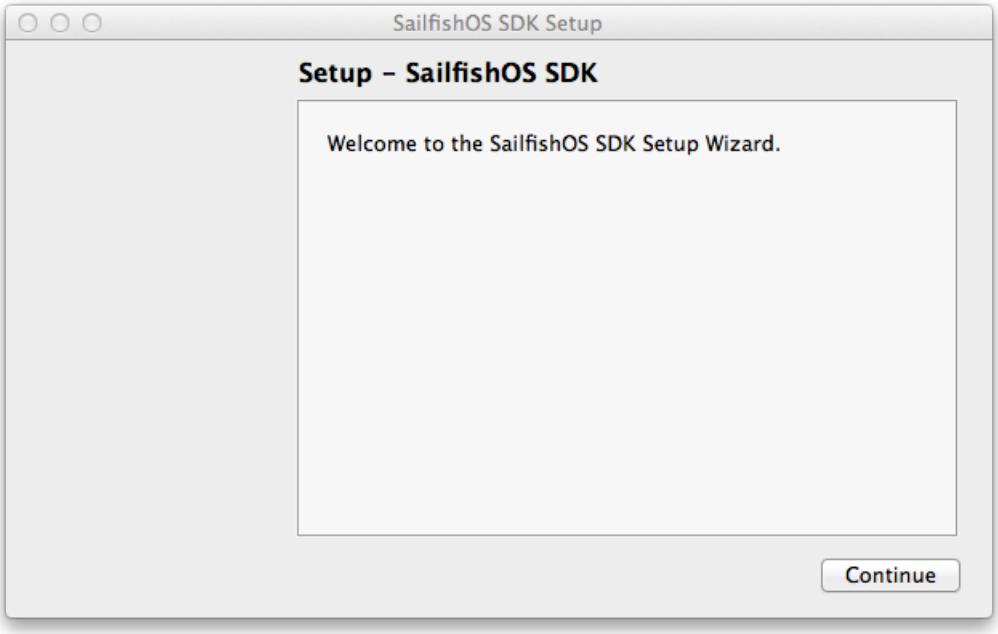


Figure 9: Install SailfishOS SDK, step 1.

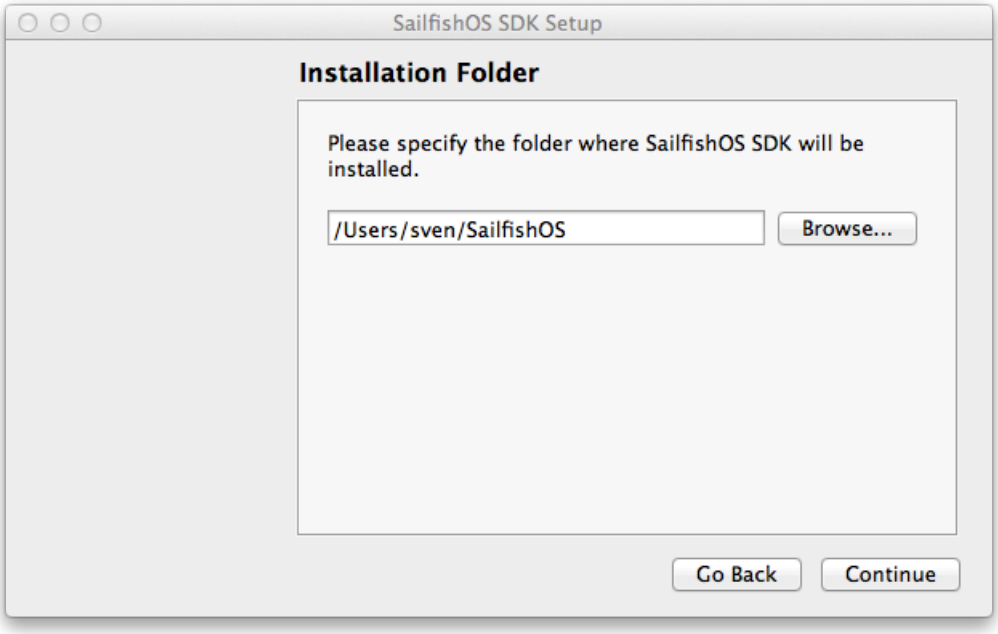


Figure 10: Install SailfishOS SDK, step 2.

Since the Alpha2 SDK you can select a separate folder for your source code if it is stored outside your home folder. Since the Alpha3 SDK this setting should work.

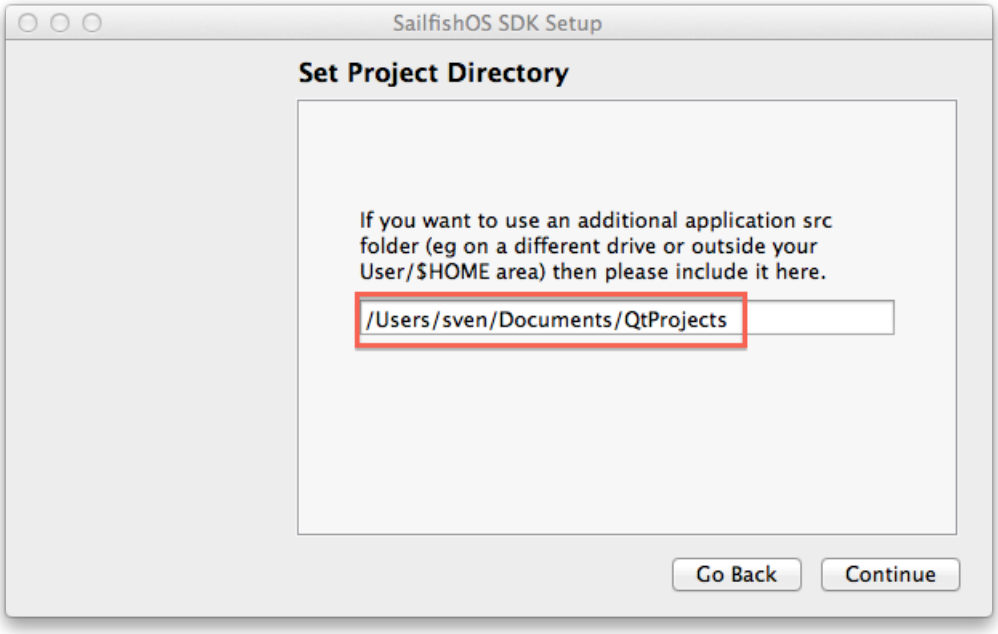


Figure 11: Install SailfishOS SDK, step 3. Enter the path to your source code. There is no folder selector dialog, you must enter it by hand.

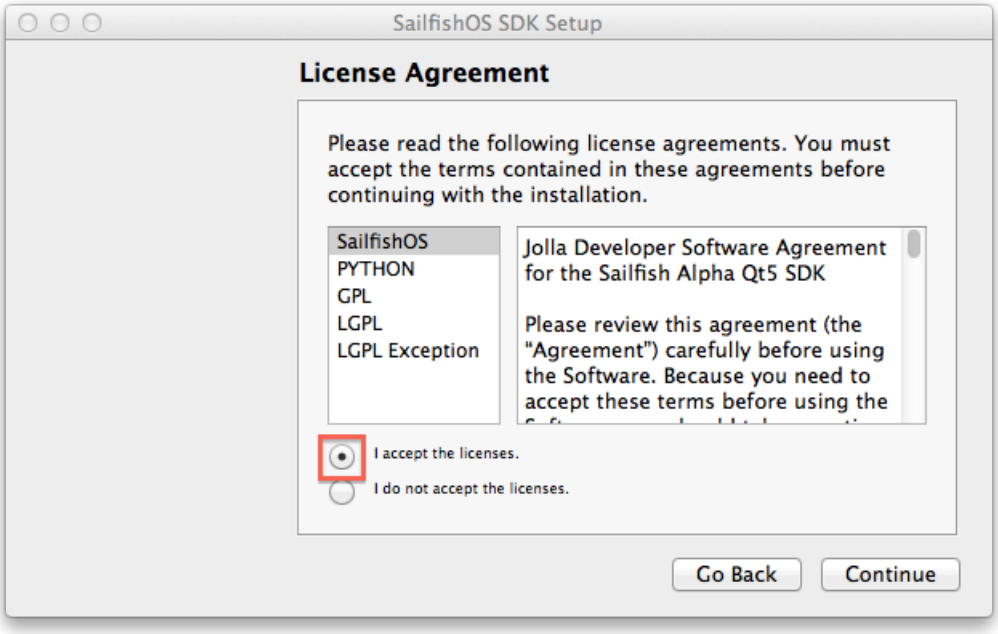


Figure 12: Install SailfishOS SDK, step 4.

The Alpha3 SDK added a screen here, where you (de-)select components.

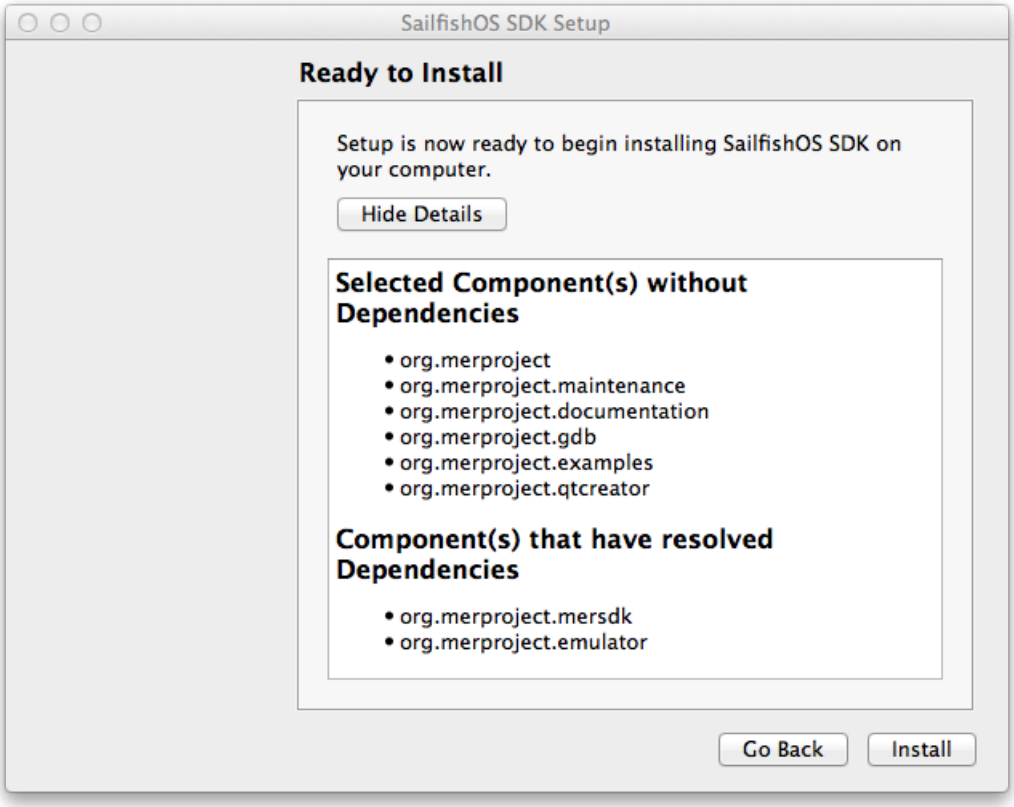


Figure 13: Install SailfishOS SDK, step 5.

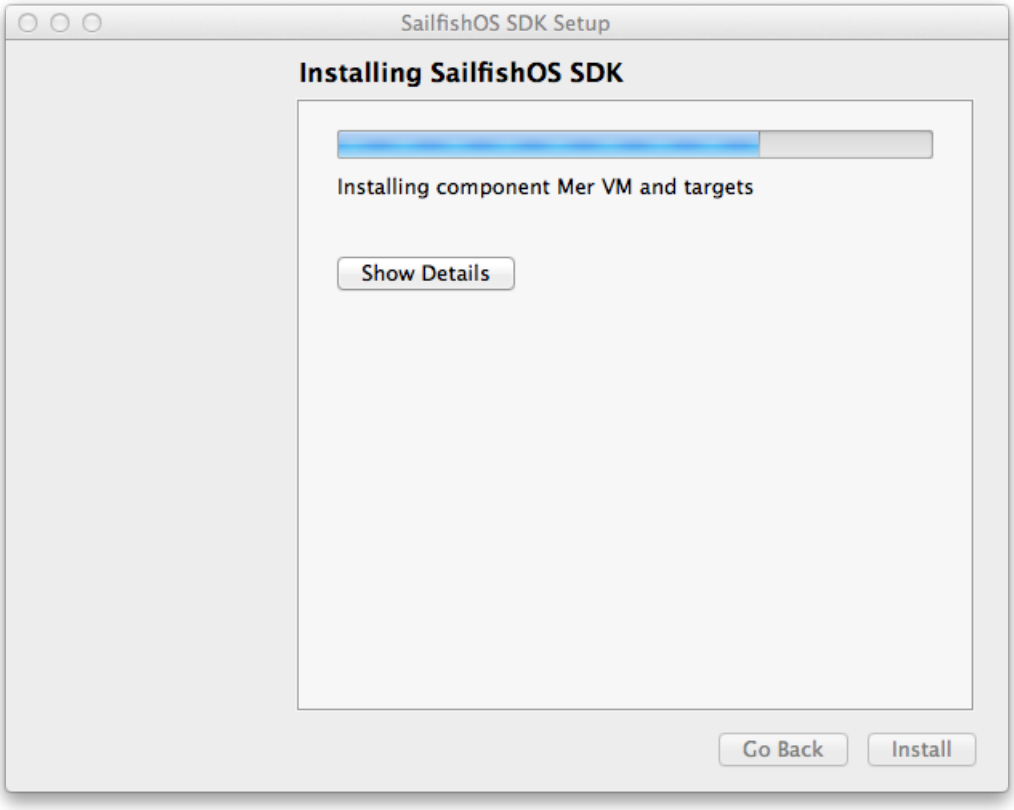


Figure 14: Install SailfishOS SDK, step 6.

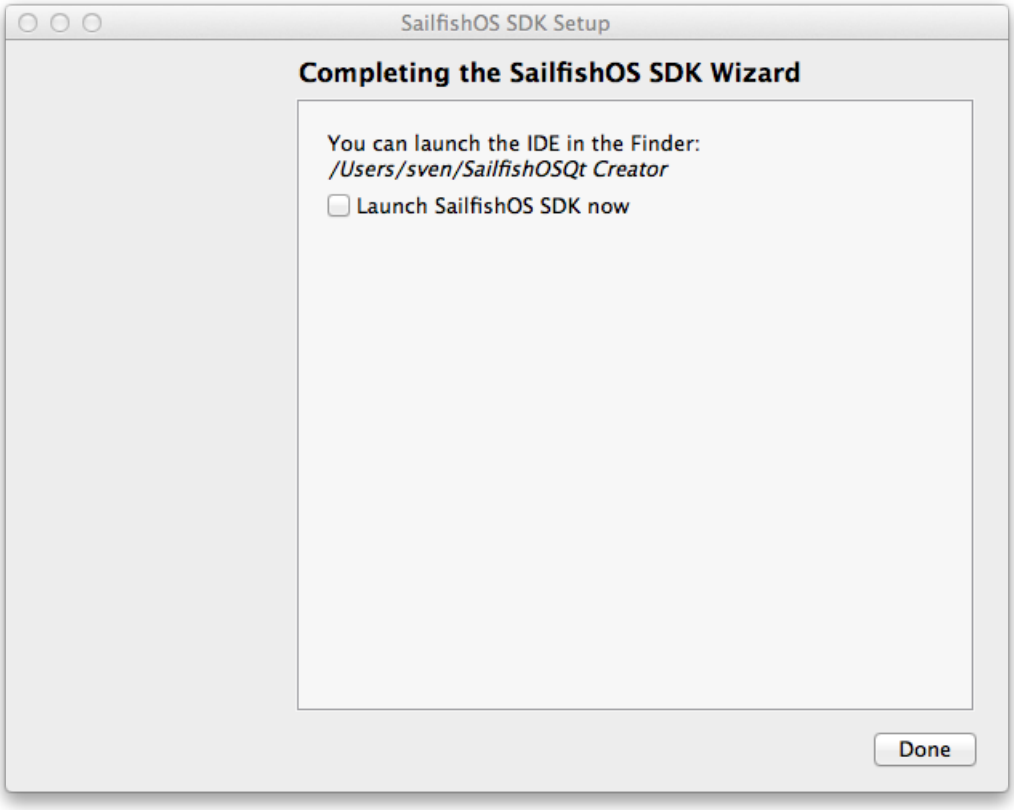


Figure 15: Install SailfishOS SDK, step 7.

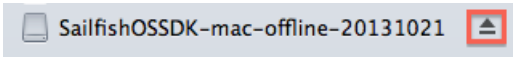


Figure 16: Unmount SailfishOS SDK disk image, step 8.

With the installation came two hidden directories, you should know about. More about those directories will follow later on.

```
$HOME/.config/SailfishAlpha3
$HOME/.scratchbox2
```

After you installed the SDK, you should immediately update the components.

Figure 17: QtCreator show that there are updates for the SDK.

As of now the progress inside Jolla is at good pace, so it might be that there is some stuff slightly out of date in the installer (see figures 18 and 19).

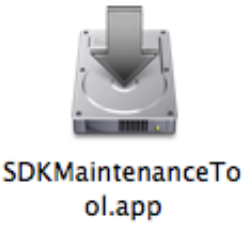


Figure 18: Inside the SailfishOS folder you find the maintenance application. Run it directly after the installation.

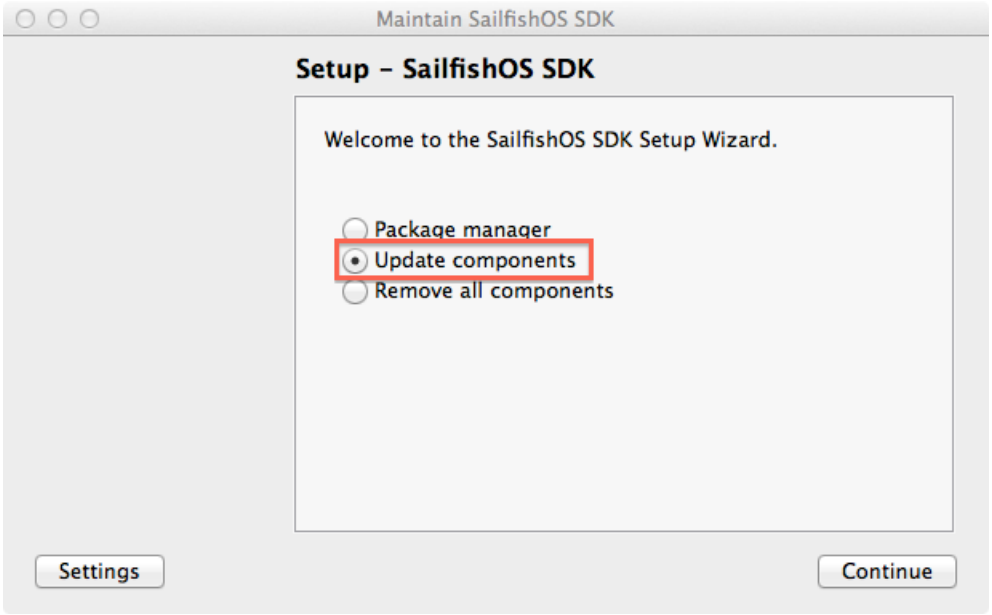


Figure 19: Update components, choose everything that is in store.

With the SDK comes QtCreator[2], a complete IDE for C++ development. This

[2]Look in in the "bin" folder of the SDK.

IDE is part of the Qt Framework[qt01] and is simply reused[3]by Jolla. Personally I use the QtCreator on my machines as well and for better differentiation I made a custom icon for the one in the SailfishOS SDK - feel free to download and use it, too[hc03].

Figure 20: Alternative icon for the QtCreator inside the SDK.

2.2 Windows

Double click the VirtualBox-4.3.4-91027-Win.exe executable that you have downloaded and follow the installer.

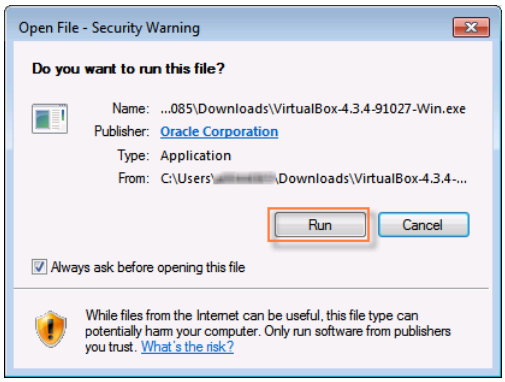


Figure 21: Install VirtualBox, Step 1.

[3]Customized with some little tweaks to suite the SailfishOS development.

Figure 22: Install VirtualBox, Step 2.

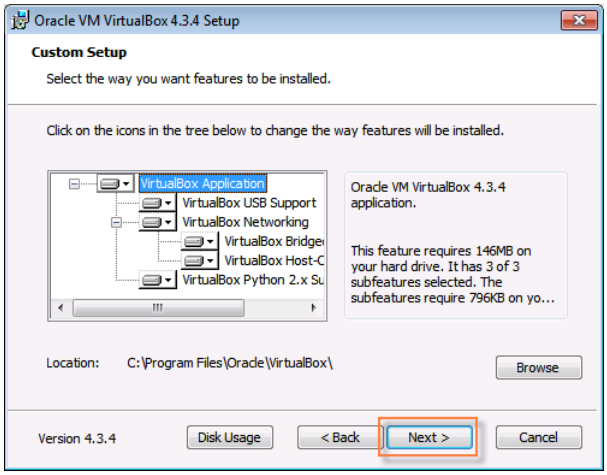


Figure 23: Install VirtualBox, Step 3.

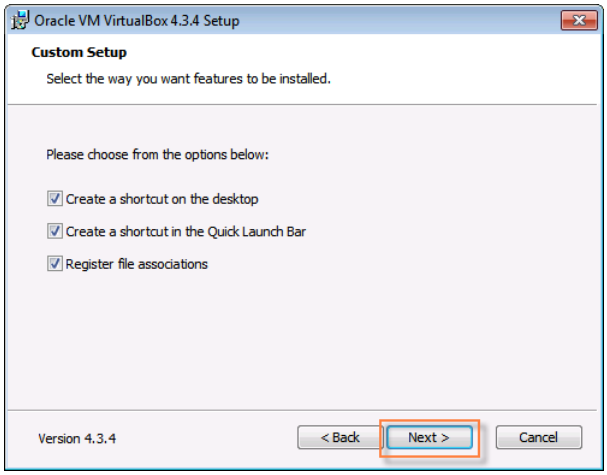


Figure 24: Install VirtualBox, Step 4.

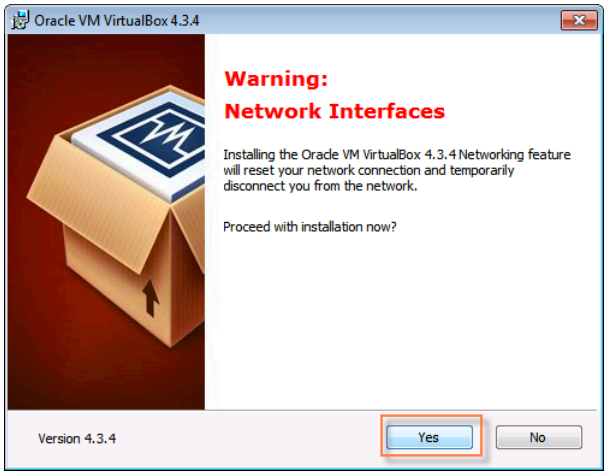


Figure 25: Install VirtualBox, Step 5.

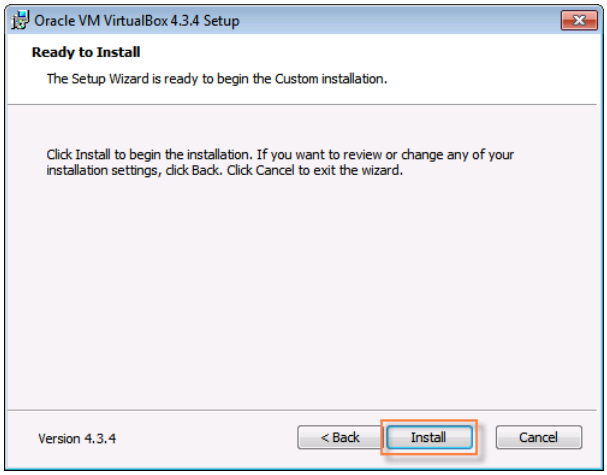


Figure 26: Install VirtualBox, Step 6.

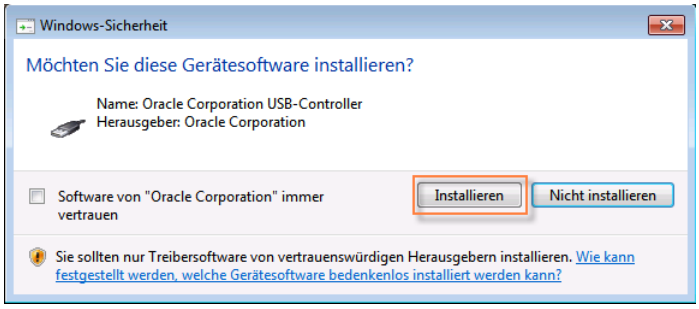


Figure 27: Install VirtualBox, Step 7.

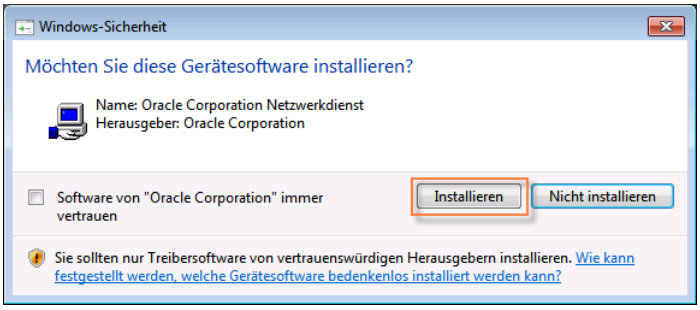


Figure 28: Install VirtualBox, Step 8.

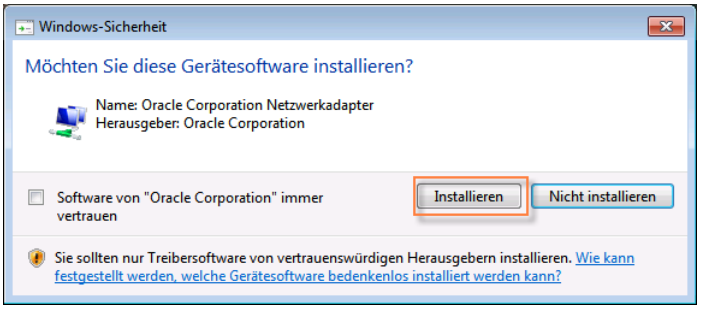


Figure 29: Install VirtualBox, Step 9.

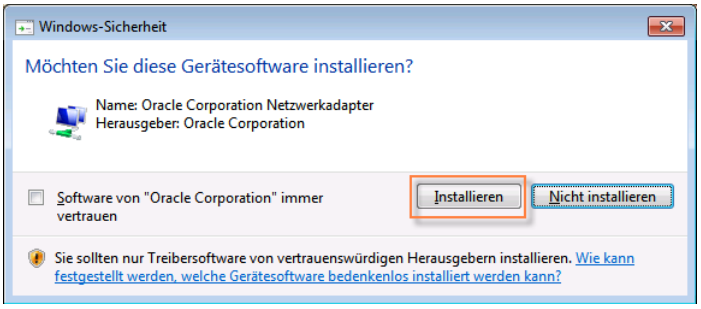


Figure 30: Install VirtualBox, Step 10.

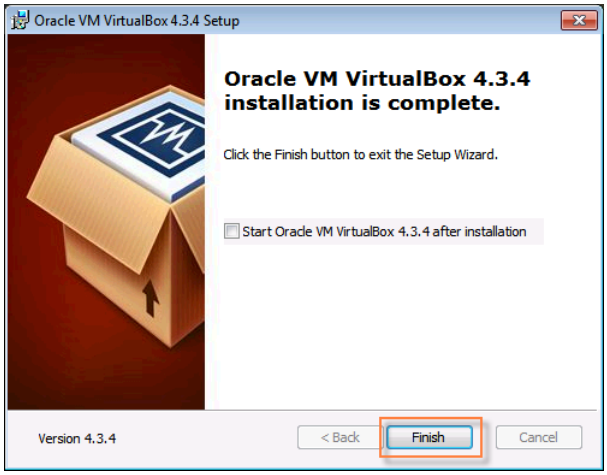


Figure 31: Install VirtualBox, Step 11.

Start the SailfishOSSDK-Alpha-1310-Qt5-windows-offline.exe executable and install the SDK.

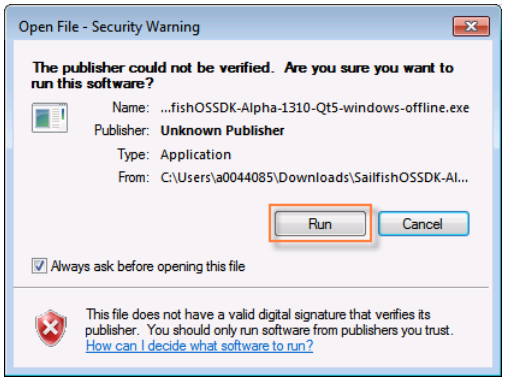


Figure 32: Install SDK, Step 1.

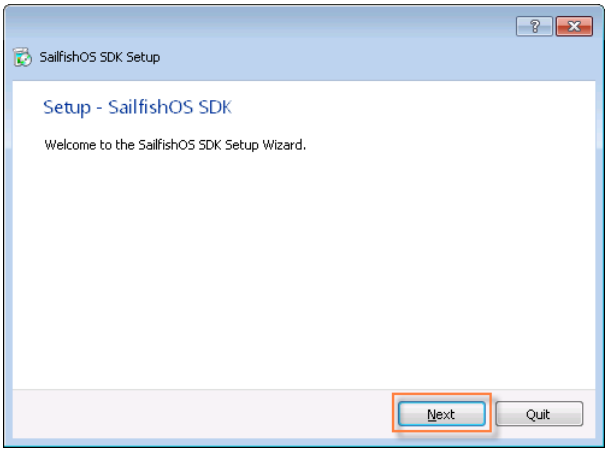


Figure 33: Install SDK, Step 2.

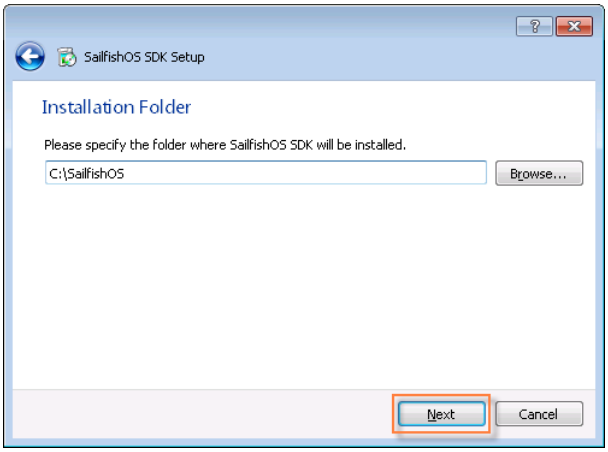


Figure 34: Install SDK, Step 3.

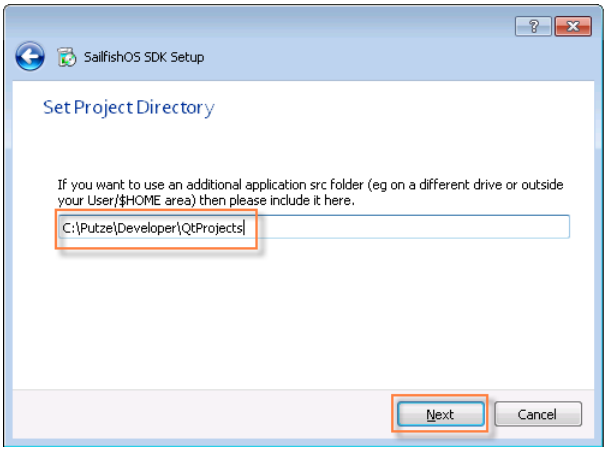


Figure 35: Install SDK, Step 4.

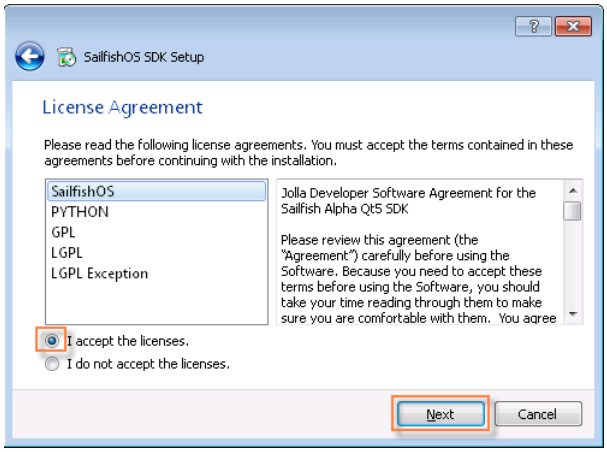


Figure 36: Install SDK, Step 5.

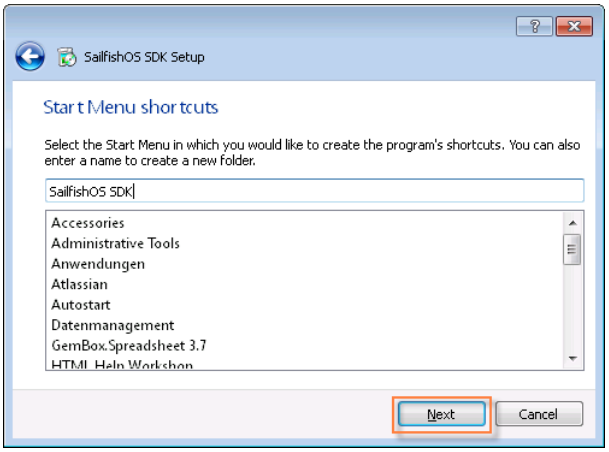


Figure 37: Install VirtualBox, Step 6.

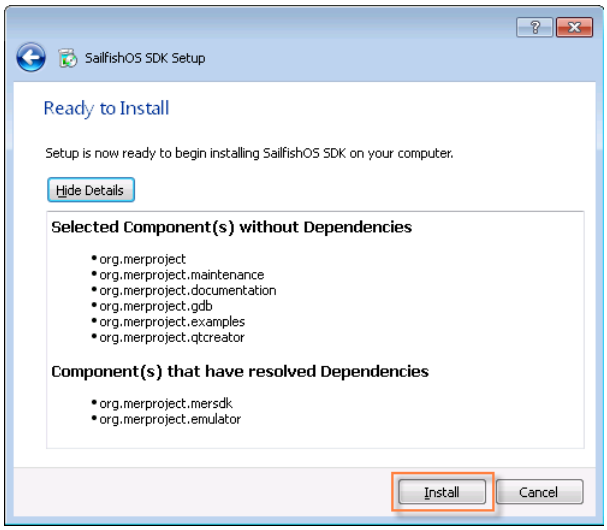


Figure 38: Install VirtualBox, Step 7.

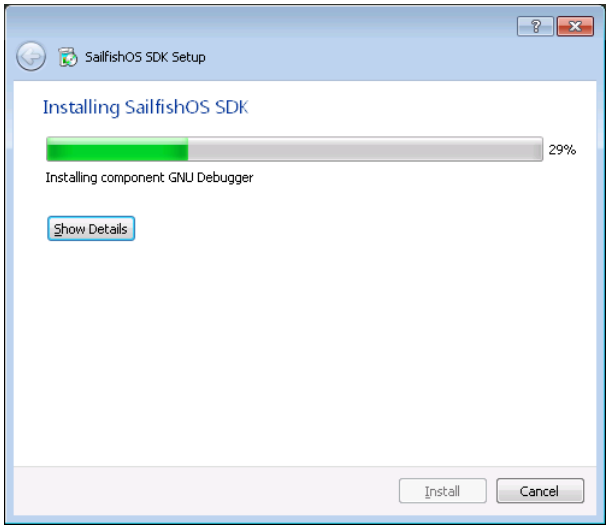


Figure 39: Install VirtualBox, Step 8.

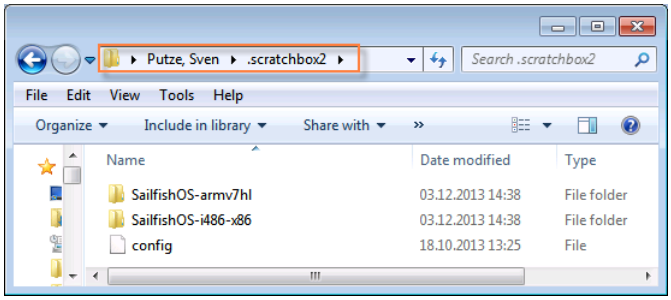


Figure 40: Install VirtualBox, Step 9.

Start the maintenance tool from the start menu or from the SailfishOS SDK directory SDKMaintenanceTool.exe and update all components.

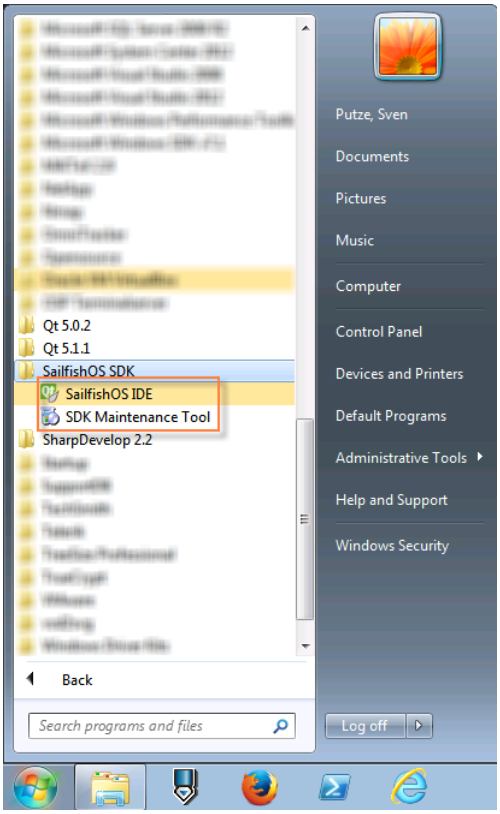


Figure 41: SDK and maintenance tool

With the installation come some directories.

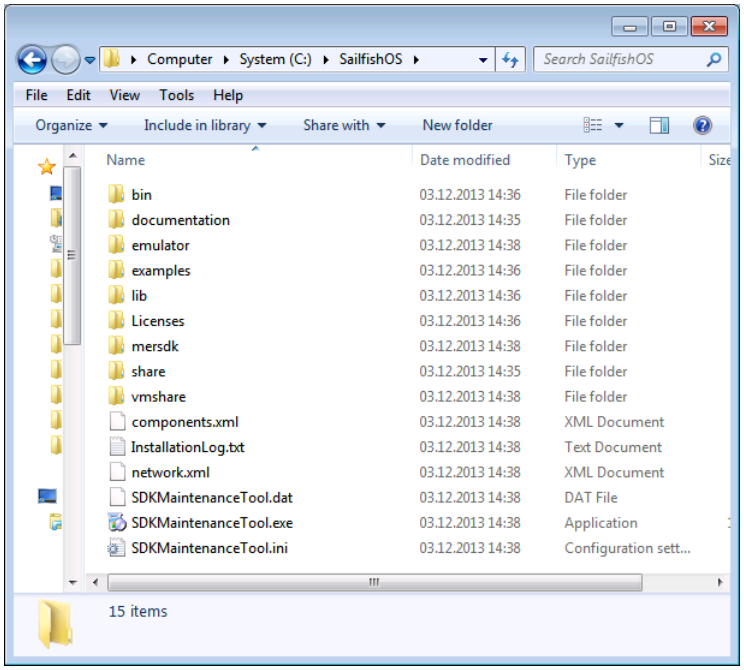


Figure 42: Directory after installation.

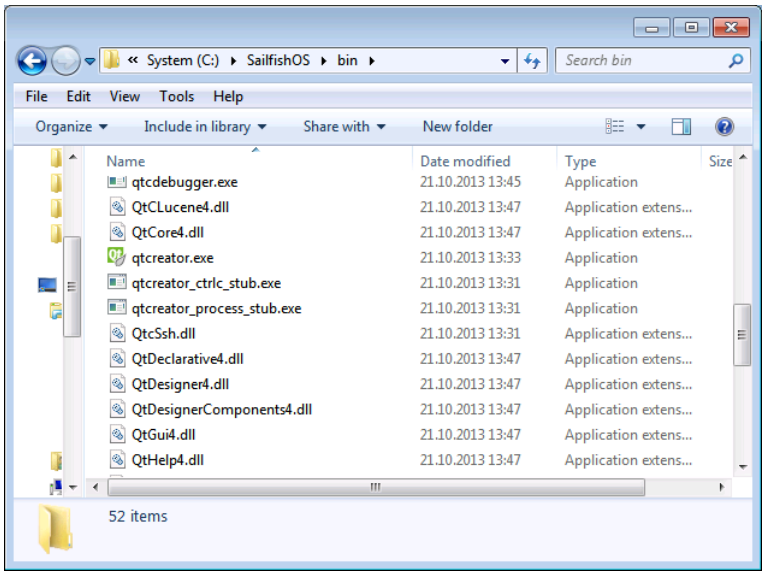


Figure 43: Directory after installation.

2.3 Linux

Download the Linux installer from the VirtualBox homepage. My distribution of choice is Debian based, so I chose the 64bit Debian package. Open your terminal and install with

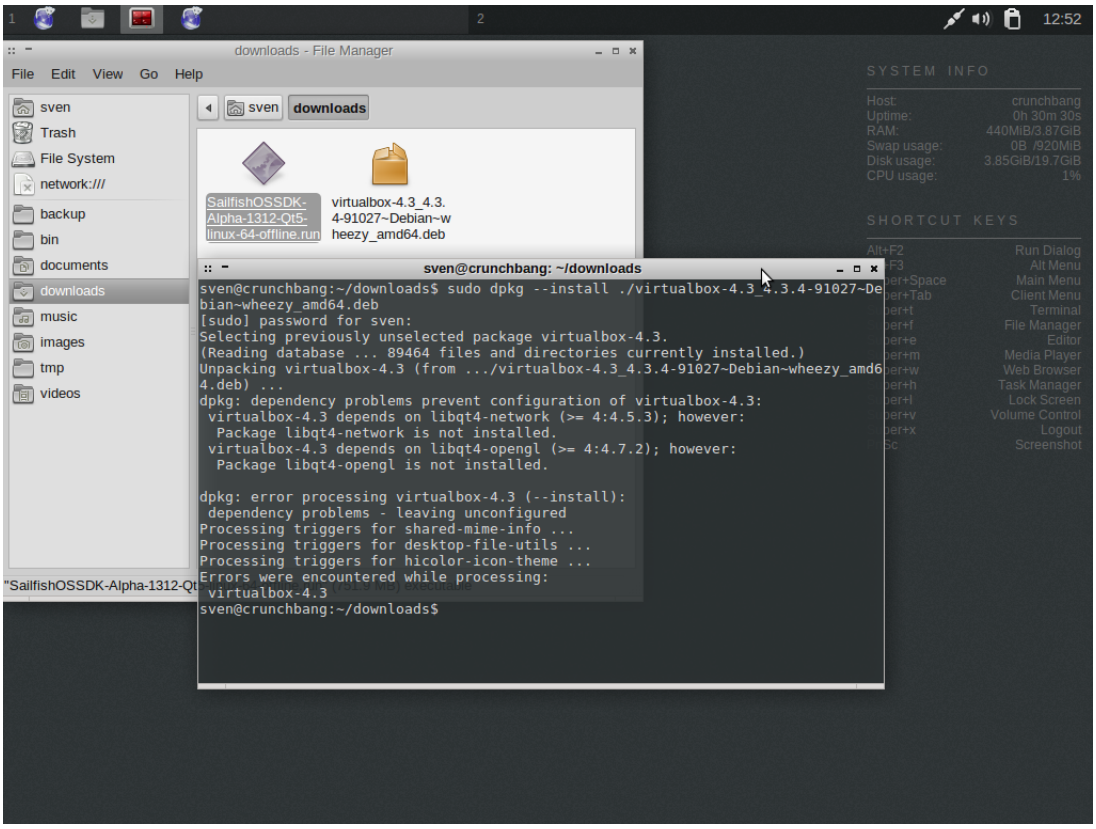


Figure 44: Install VirtualBox from the terminal.

```
cd downloads
sudo dpkg --install ./virtualbox-4.3_4.3.4-91027~Debian~wheezy_amd64.
    deb
# as you can see in the screenshot there are missing dependencies,
    install them with
sudo apt-get -f install
# and now finnish the installation
sudo dpkg --install ./virtualbox-4.3_4.3.4-91027~Debian~wheezy_amd64.
    deb
```

Now download the installer from [sailfishos01],

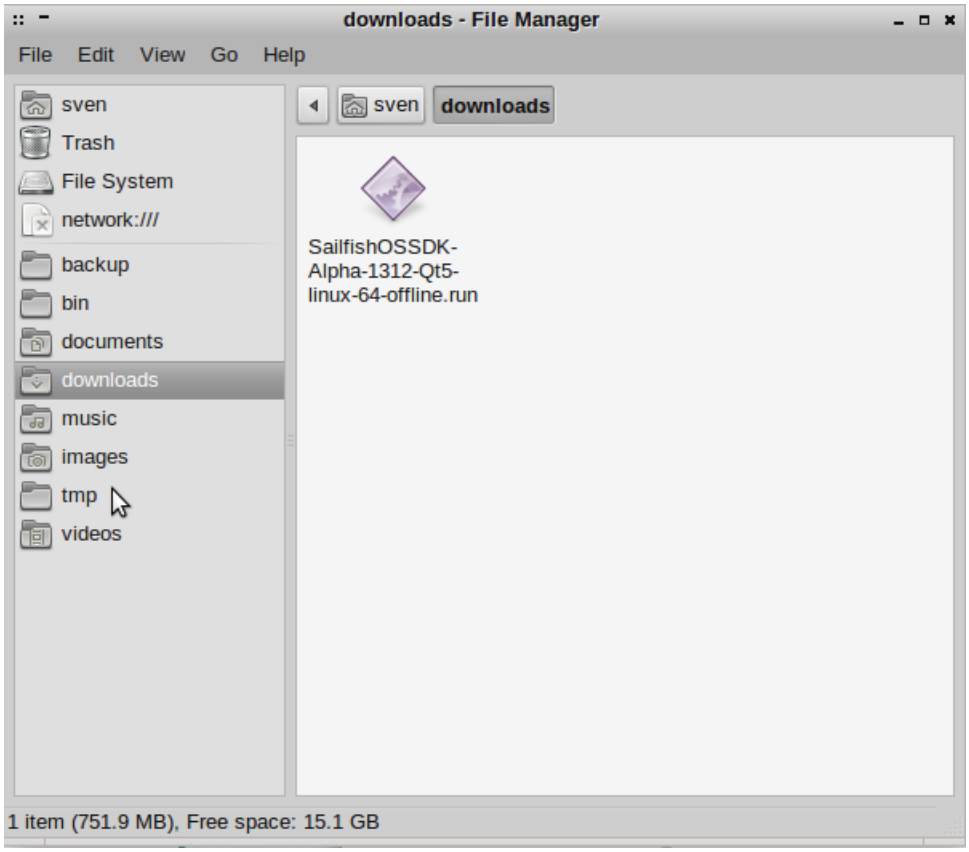


Figure 45: Install SailfishOS SDK on Linux, step 1.

make it executable and start it. From now on it's mostly like on OSX and/or Windows.

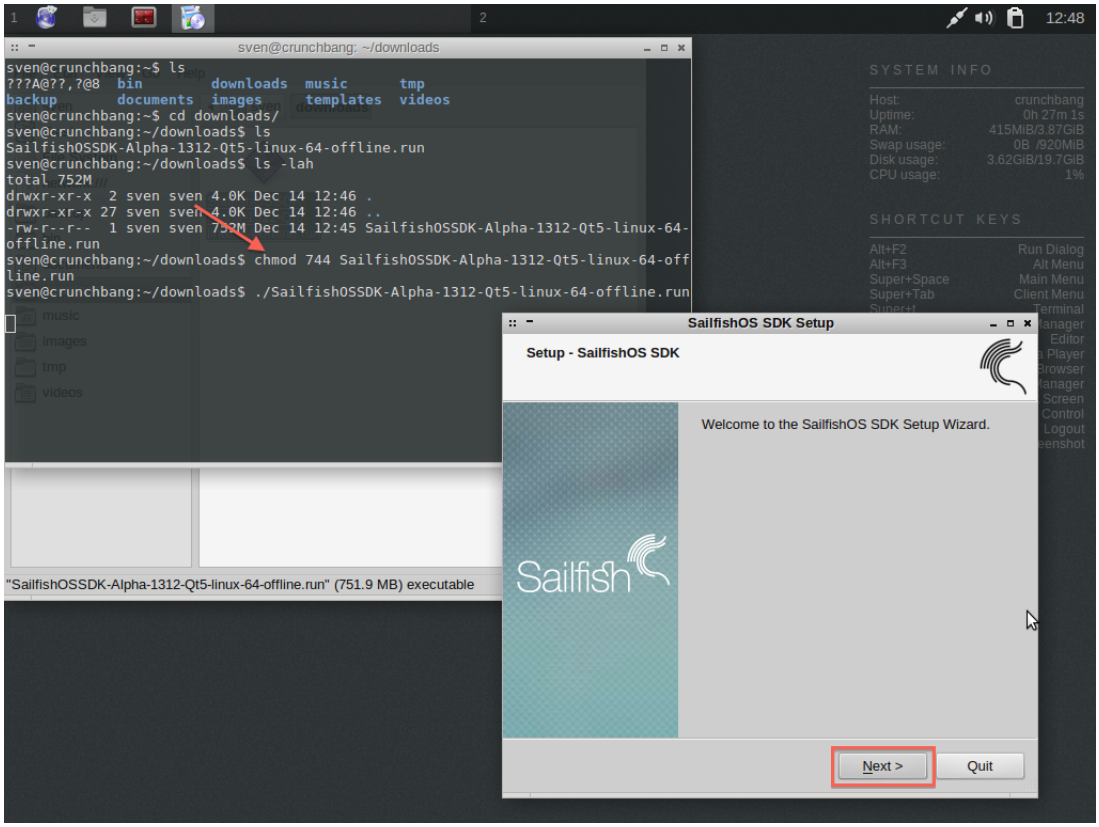


Figure 46: Install SailfishOS SDK on Linux, step 2.

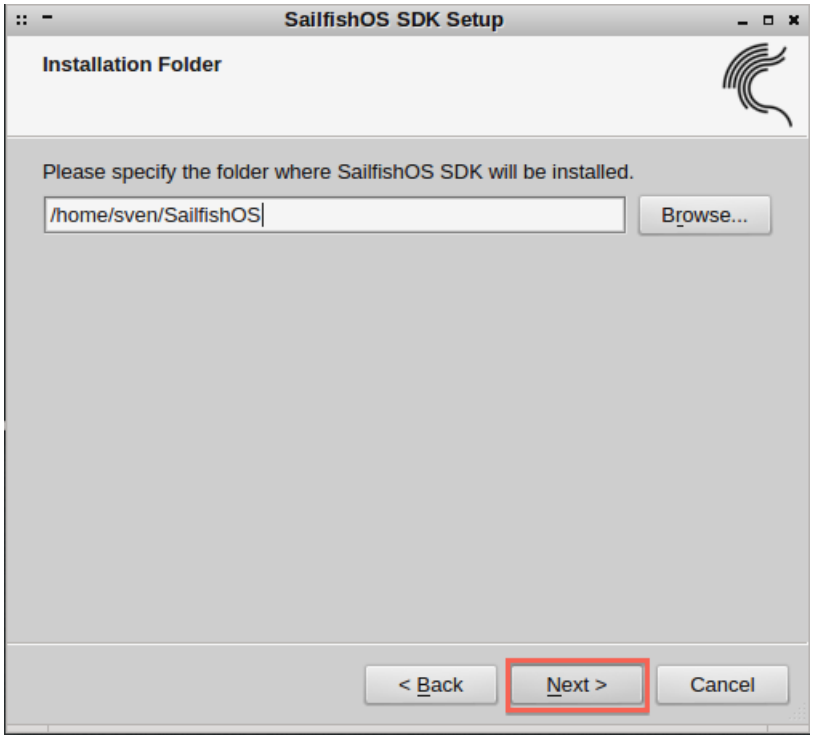


Figure 47: Install SailfishOS SDK on Linux, step 4.

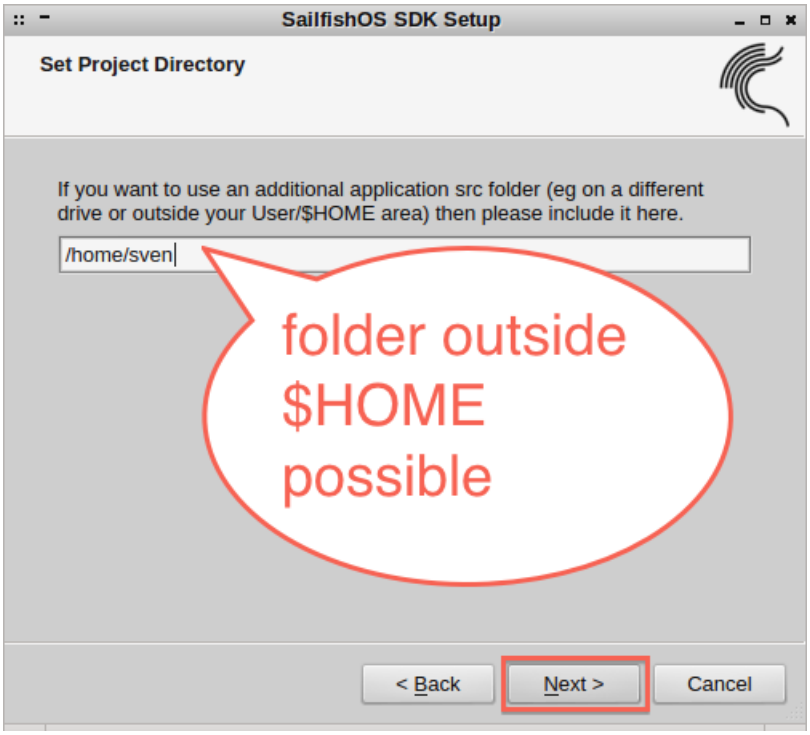


Figure 48: Install SailfishOS SDK on Linux, step 5.

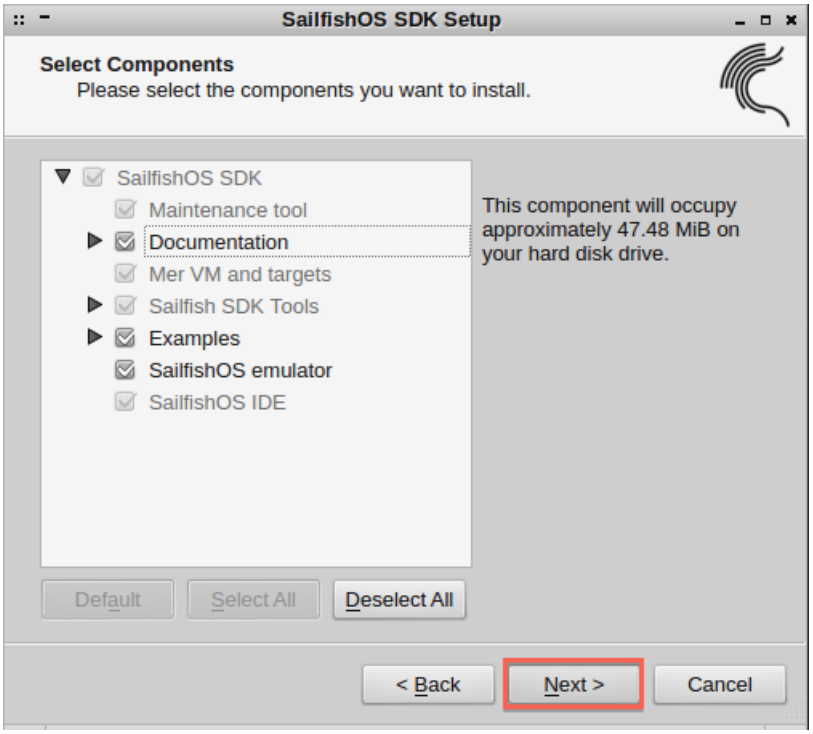


Figure 49: Install SailfishOS SDK on Linux, step 6.

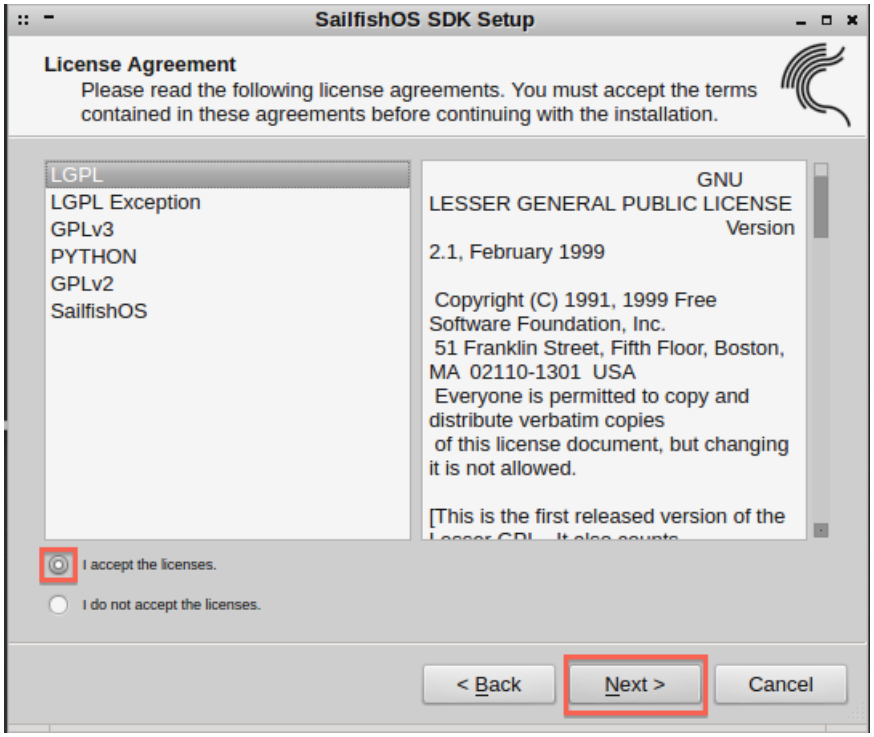


Figure 50: Install SailfishOS SDK on Linux, step 7.

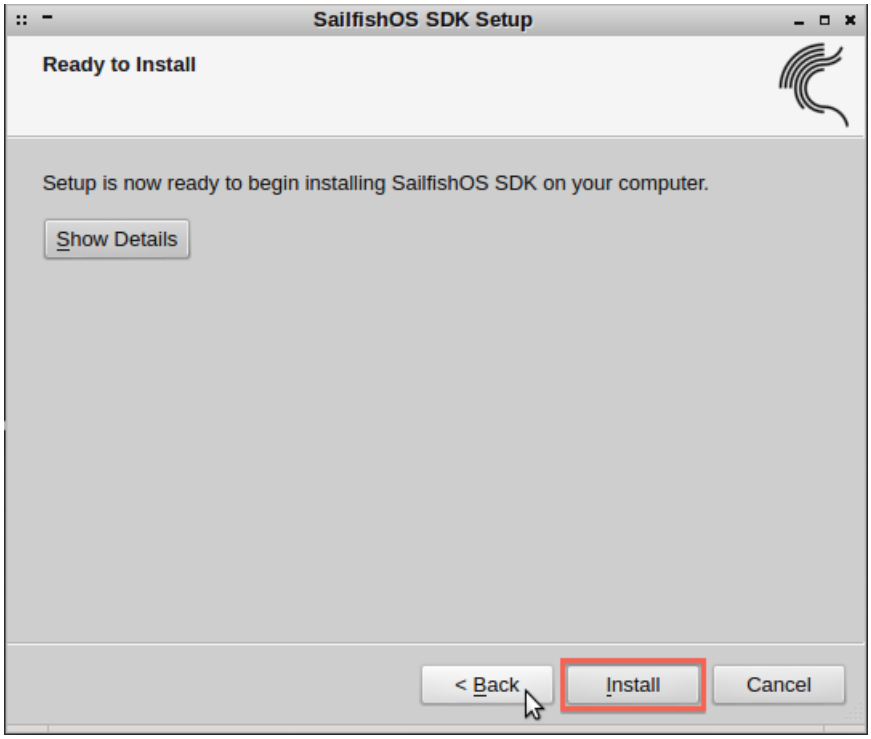


Figure 51: Install SailfishOS SDK on Linux, step 8.

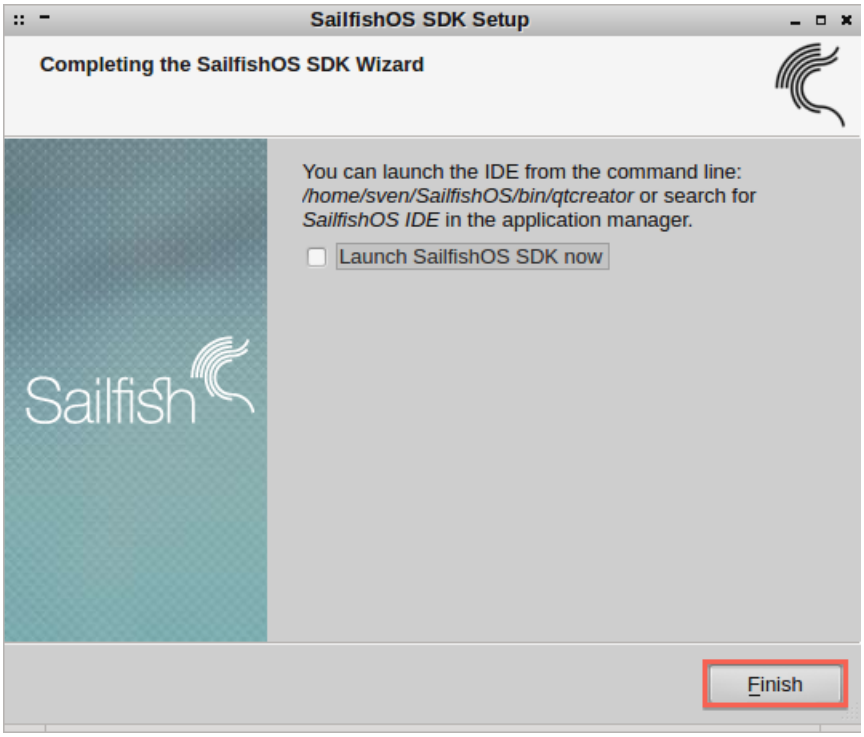


Figure 52: Install SailfishOS SDK on Linux, step 9.

Now you can start and use the SDK. How to start depends on your distribution. Maybe a menu entry was created, maybe you must create one on your own. The binary is in the `~/SailfishOS/bin` folder[4].

2.4 Remove plugins

If you want to improve the startup time of QtCreator, you can deactivate plugins you don't need or want[5]. Just don't shoot in your foot here.

[4] If you have chosen the default location, of course.
[5] On Windows and Linux the plugins should be found in Extras/Plugins.

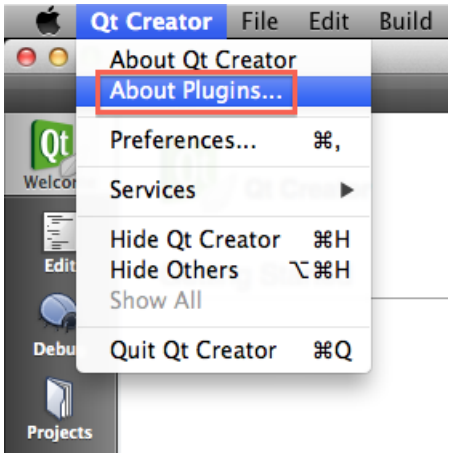


Figure 53: About plugins = manage plugins.

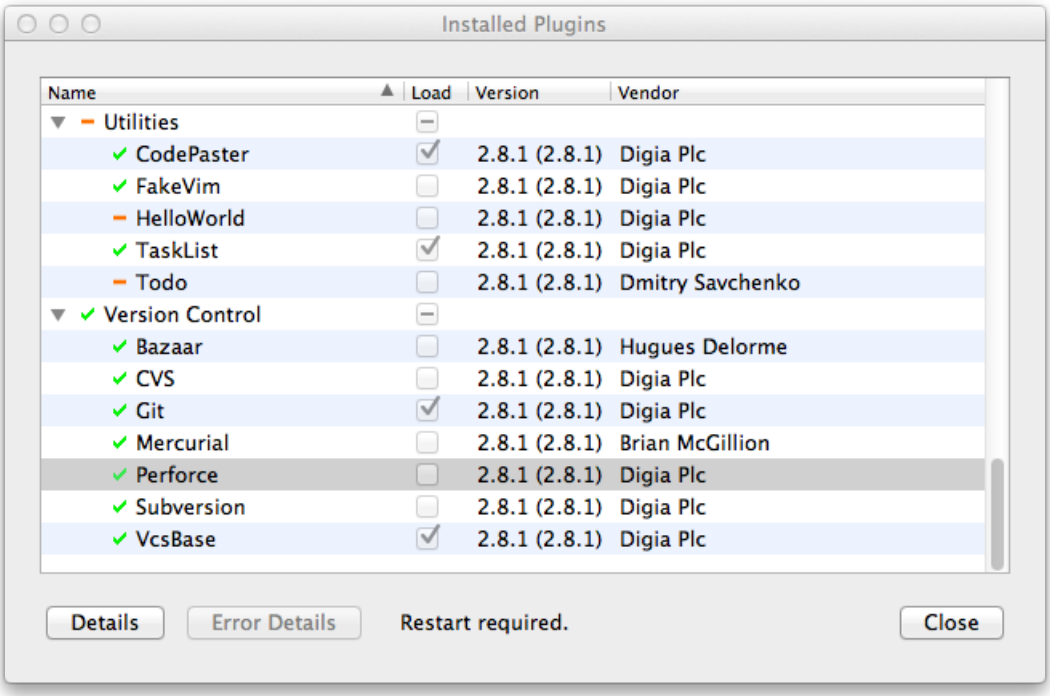


Figure 54: Deactivate every plugin you don't need.

3 Quickstart

Start the QtCreator from the fresh installed SDK.

"The SDK comes with a handy SailfishOS application template that gives you a quick way to create your very first Sailfish OS application. Just go to File-> New File or Project in the IDE"[sailfishos2]

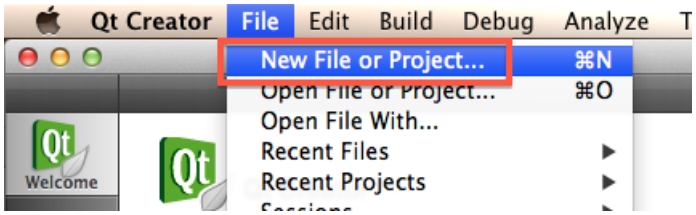


Figure 55: First example, step 1.

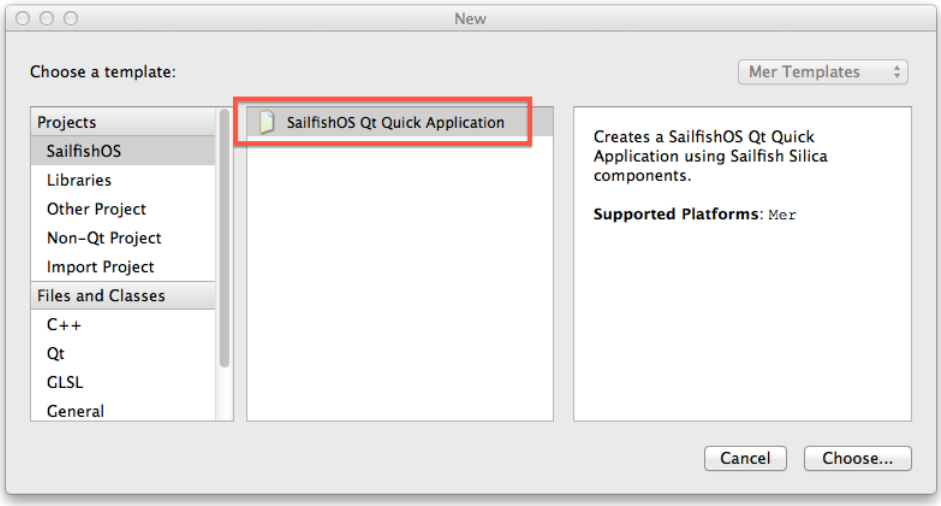


Figure 56: First example, step 2.

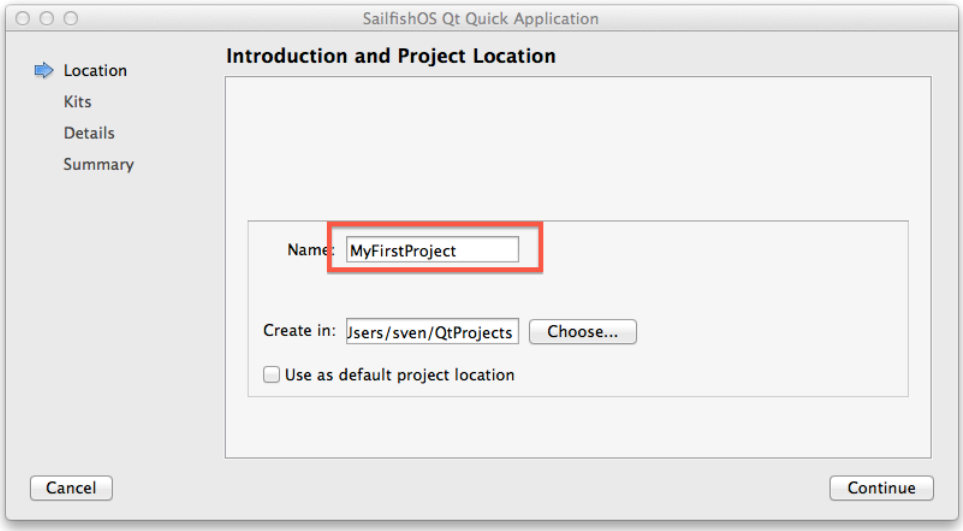


Figure 57: First example, step 3.

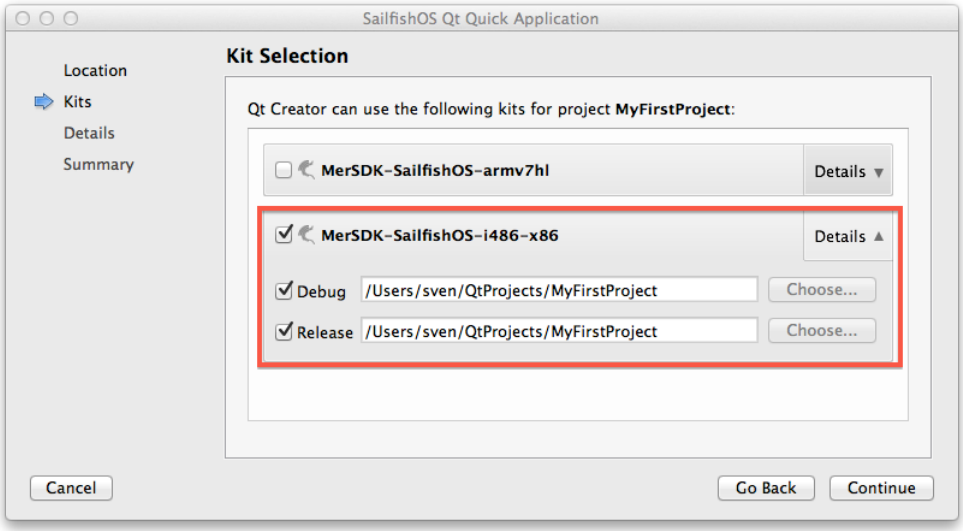


Figure 58: First example, step 4.

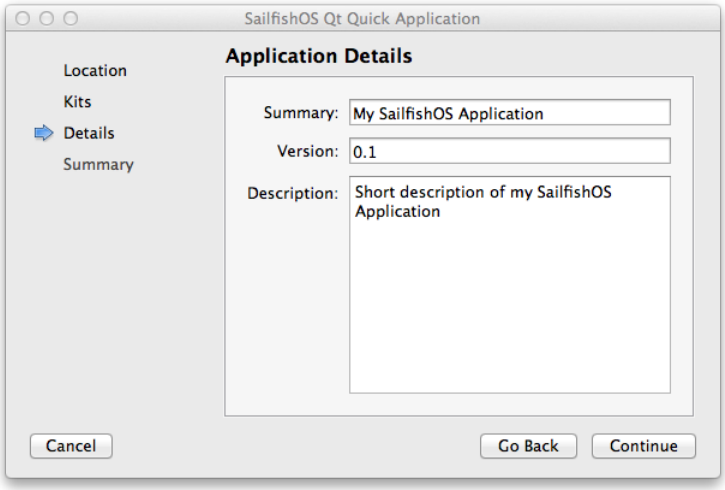


Figure 59: First example, step 5.

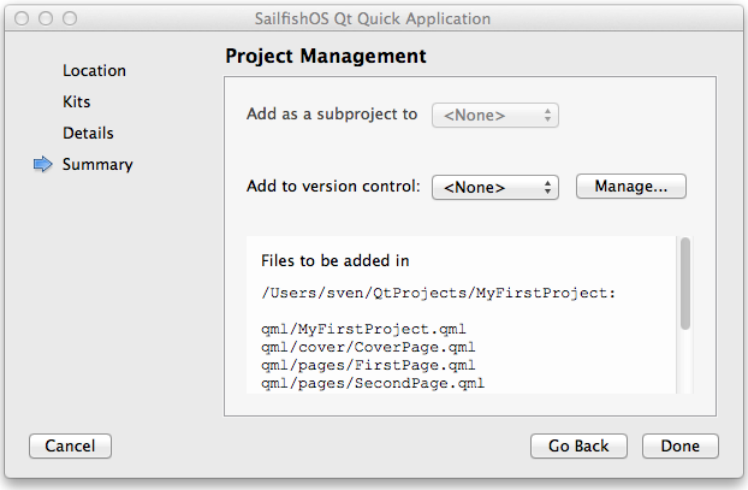


Figure 60: First example, step 1.

Start the SDK from inside the QtCreator.

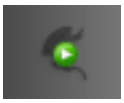


Figure 61: Starting the SDK.

When the virtual machine with the SDK is running, apply updates if necessary.

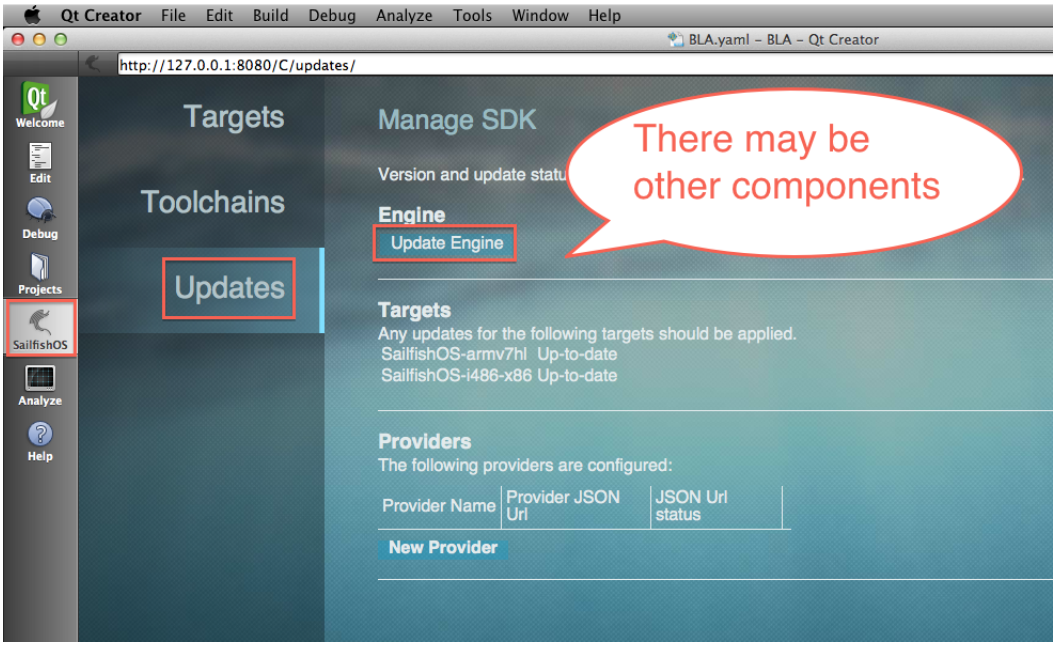


Figure 62: Updating the SDK.

Start the Emulator.

Figure 63: Starting the Emulator.

Compile and run your application.

Figure 64: Build and run the application.

Easy as pie, also see figure 100 on page 81. But what happens when and where if you click on those Icons? Look in section *Know your tools*.

A good starting point to look for a more sophisticated starting example is *The missing HelloWorld. Wizard included* by Artem Marchenko[gh01]. You should check that out! It has all the tweaks needed to bring your app to the Jolla store!

And you should look in the `examples` folder of the SDK of course.

4 Know your tools

Jolla chose the Qt framework to be part of their technology stack. “Qt is a cross-platform application and UI framework for developers using C++ or QML, a CSS & JavaScript like language. Qt Creator is the supporting Qt IDE. Qt, Qt Quick and the supporting tools are developed as an open source project governed by an inclusive meritocratic model. Qt can be used under open source (LGPL v2.1) or commercial terms.”[qt01]

“Qt - code once, deploy everywhere”, that’s the mantra of the Qt framework. If you have developed for more than one platform in the past, you know that this sounds like heaven. Maintaining different source code and technologies for each and every platform is a tedious task that can eat up all your developer resources. As if software development is not difficult enough if you stay on one platform[6].

So there were good reasons to choose Qt as framework, no doubt about that. As of now you can develop for Android, iOS, BlackBerry and of course SailfishOS. If you look into the documentation and examples of all these platforms, you will find that those examples assume, that you are developing for this platform only. Quite stupid, if you use the Qt framework. Understandable if you think about the effort that would be necessary to build a documentation that incorporates all other possible platforms. For some time now I was wondering how I should organize my code in such a way, that allows me to develop for more than one target platform at a time. It’s not just compiling for another platform! Each platform has a unique UI that behaves in an own different way, e.g. SailfishOS is gesture based, other ones are touch based. In the long run you will create an UI for each of those targets. Period. Patterns like MVC[wiki01] will come to mind, using separate business logic, yada yada.

To cut a long story short, why am I writing about this stuff, this section is supposed to be about tools? When you prepare a software project for the use for more than one target platform, you will start organizing stuff differently. Maybe you use folders that have the name of the targets to differentiate stuff that’s platform dependent. Maybe you even create a business logic that is really unique and capsulated in such clever way that it can be reused and does not know anything about the outside world. Such a business logic or model can be driven from tests, command line tools, web or different native UIs. Would be nice to have it in a separate folder or even subproject. If you start to move and/or rename things, your tools will break. Intentionally.

By examining those fractures you can learn a lot about your tools that otherwise work so silently in the background. So go on and break your tools![7]

[6] In my eyes software development is an art of craftsmanship and can not be done by Mr. Average and thus is a more or less complicated thing to do.

[7] Ok, not so short :-)

Here is what I've learned so far.

4.1 Technology stack

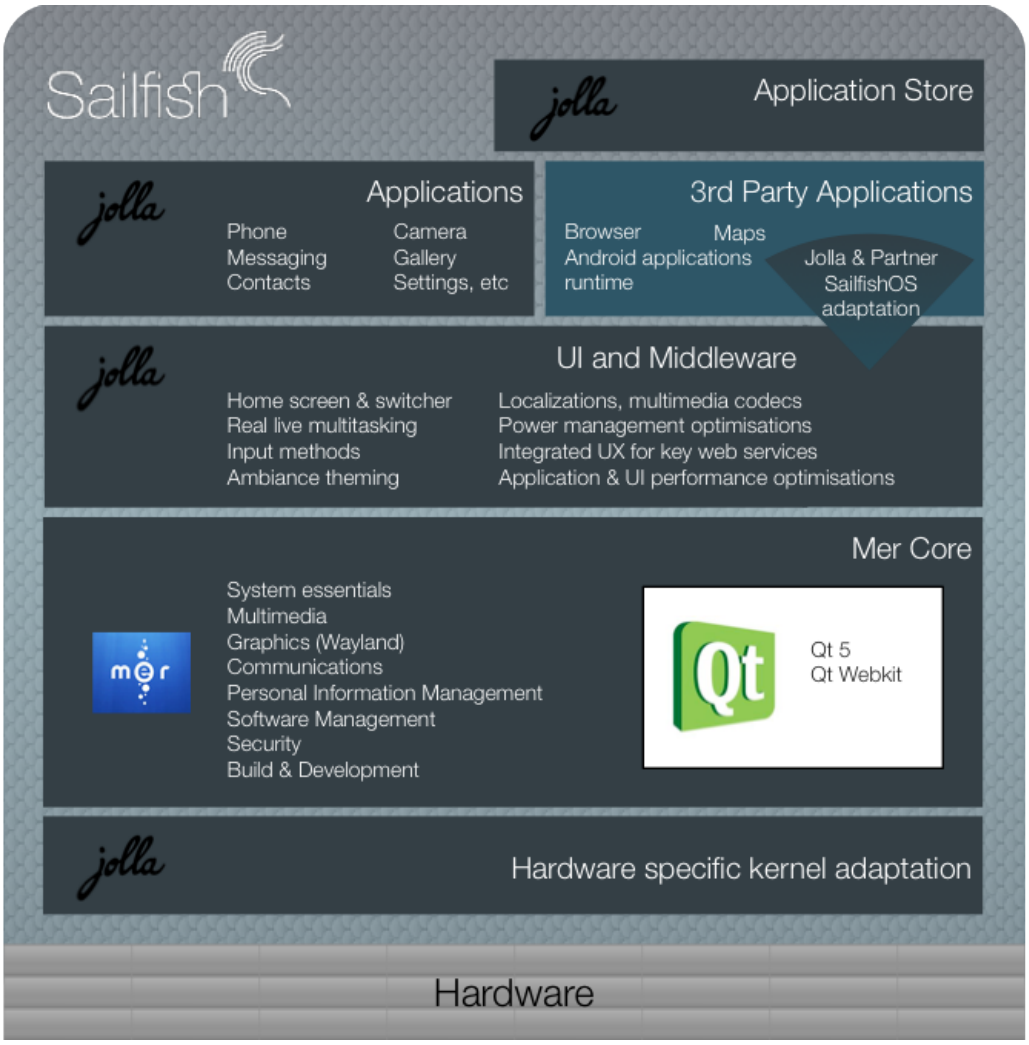


Figure 65: Sailfish architecture, taken from https://sailfishos.org/images/Sailfish_Architecture.png.

4.2 QtCreator integrated development environment (IDE)

"QtCreator is a cross platform integrated development environment (IDE) tailored to the needs of Qt developers. It has been extended to add support for Sailfish UI application development using Sailfish Silica components. It provides a sophis-

ticated code editor with version control, project and build management system integration."[sailfishos3].

Reusing an existing open source IDE is a smart move from Jolla. Why should they waste resources on developing something that has already done by others? Or why should they burn up their staff for all those development solutions out there? Be it Visual Studio, Eclipse, Emacs or even Vi. If you really dive in the tools, you can also use those but I doubt that Jolla will provide you with support if something does not work. Working with QtCreator is also quite natural in the Qt universe albeit being a fast IDE. So have a look in the preferences[8].

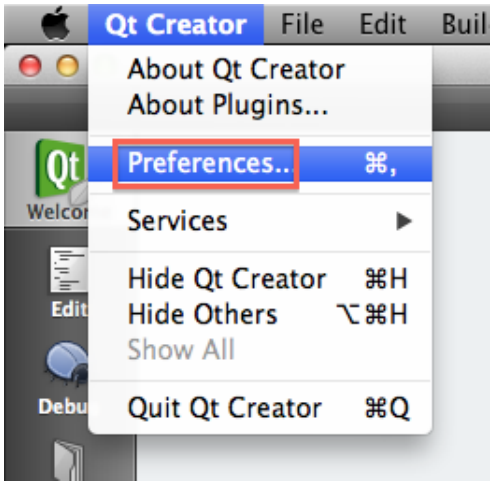


Figure 66: Open the QtCreator preferences.

I will not walk through every setting of QtCreator, [qt02] is a better place to start for basic questions. Also I will not use the order of tabs in the preferences, but try to follow the sequence in which those tools touch your source code.

4.2.1 kits

But before we do that, we must talk about *kits*. A kit is kind of an umbrella setting, which combines the information of the following bits and pieces, like `qmake`, `compiler`, and `device type`. This is the information hub that QtCreator uses to pull all information together and initiate its actions when you build or start your application.

[8] On Windows and Linux they should be found in Extras/Options.

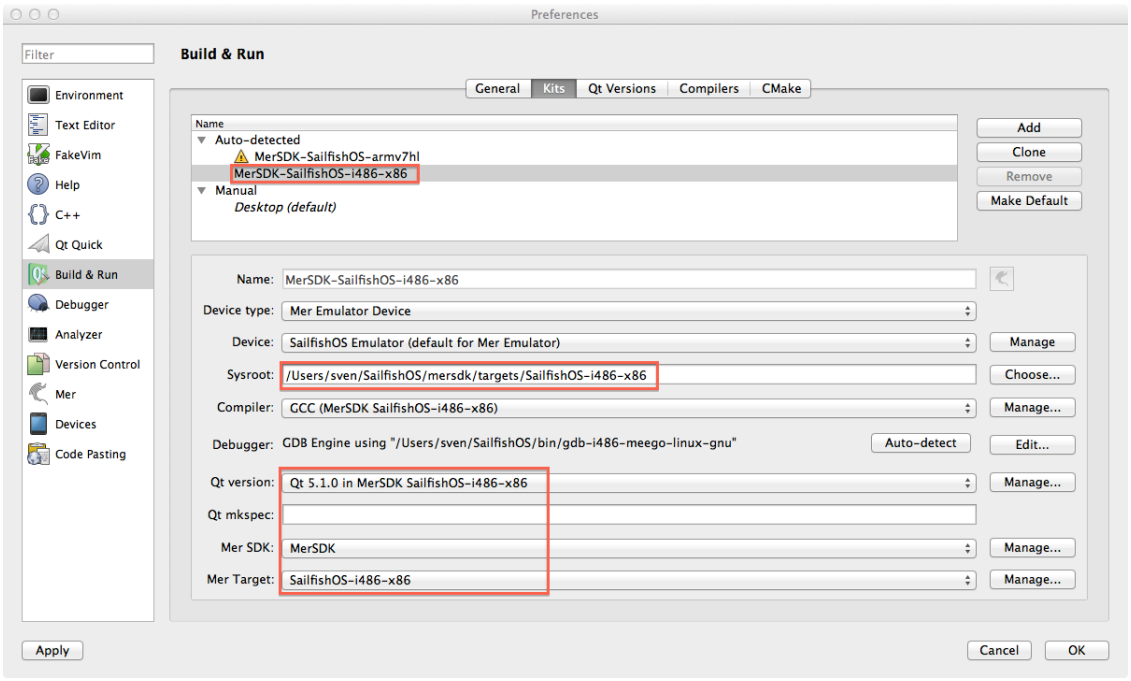


Figure 67: Preferences, kits tab.

So far there are 2 kits defined:

- MerSDK-SailfishOS-armv7hl
  this kit is marked with a warning sign in the Alpha2 SDK, there is no device assigned yet.
  The Alph3 SDK brings device support but it's not configured yet.
- MerSDK-SailfishOS-i486-x86

Should you have a kit named `Desktop`, it's of no use inside the QtCreator that comes with the SailfishOS SDK, try not to build your app with this kit!

4.2.2 qmake

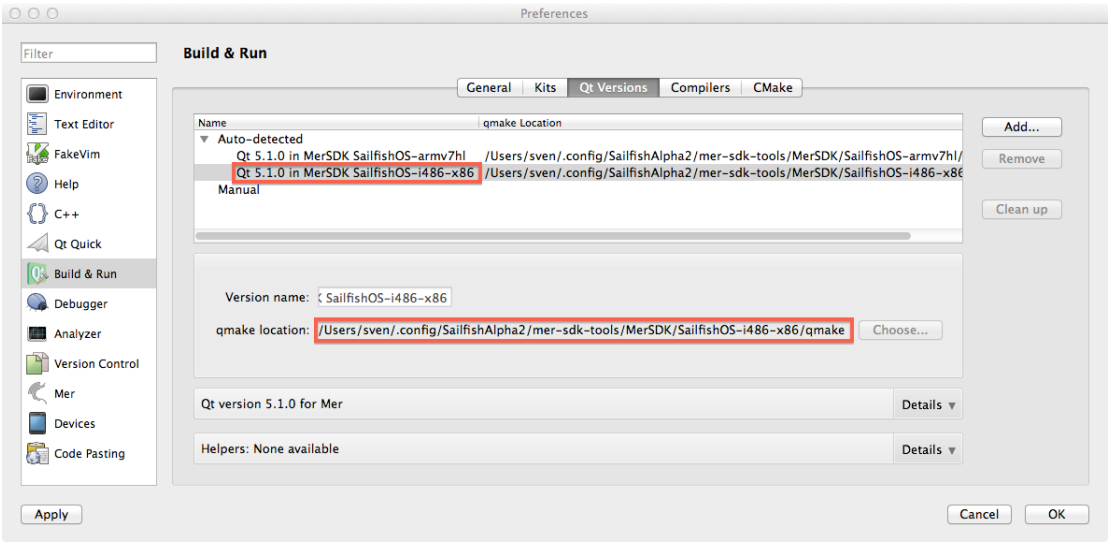


Figure 68: Preferences, QtVersions tab.

"qmake is a tool that helps simplify the build process for development project across different platforms. qmake automates the generation of Makefiles so that only a few lines of information are needed to create each Makefile. qmake can be used for any software project, whether it is written in Qt or not. qmake generates a Makefile based on the information in a project file. Project files are created by the developer, and are usually simple, but more sophisticated project files can be created for complex projects. qmake contains additional features to support development with Qt, automatically including build rules for moc and uic. qmake can also generate projects for Microsoft Visual studio without requiring the developer to change the project file."[qt03]

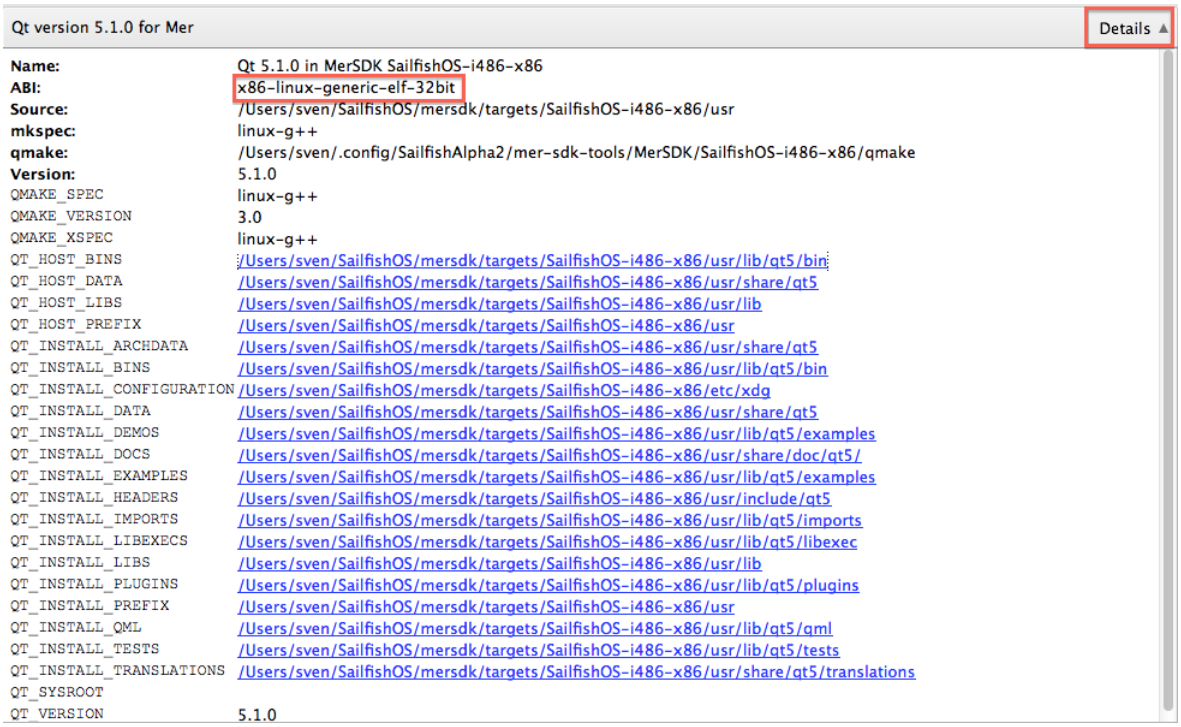


Figure 69: Preferences, QtVersions tab., qmake details

UIC is a tool that creates C++ classes from XML information generated with the UI designer inside QtCreator. This designer is for Qt widgets which should not be used with SailfishOS and is not further explained in this document.

MOC is the Meta-Object Compiler which “reads a C++ header file. If it finds one or more class declarations that contain the `Q_OBJECT` macro, it produces a C++ source file containing the meta-object code for those classes.”[qt04]. If you use `qmake` to produce your Makefile, you don’t have to worry about it, the rules are created automatically.

```
SailfishOS-i486-x86 — bash — 80×24
lastenhuone:~ sven$ cd .config/SailfishAlpha2/mer-sdk-tools/MerSDK/SailfishOS-i4
86-x86/
lastenhuone:SailfishOS-i486-x86 sven$ ls -lah
total 64
drwxr-xr-x  10 sven  staff   340B 20 Nov 14:23 .
drwxr-xr-x   4 sven  staff   136B 20 Nov 14:23 ..
-rwxrwx---   1 sven  staff   224B 20 Nov 14:23 deploy
-rwxrwx---   1 sven  staff   221B 20 Nov 14:23 gcc
-rw-r--r--   1 sven  staff    21B 20 Nov 14:23 gcc.dumpmachine
-rwxrwx---   1 sven  staff   234B 20 Nov 14:23 gdb
-rwxrwx---   1 sven  staff   222B 20 Nov 14:23 make
-rwxrwx---   1 sven  staff   223B 20 Nov 14:23 qmake
-rw-r--r--   1 sven  staff   1,9K 20 Nov 14:23 qmake.query
-rwxrwx---   1 sven  staff   221B 20 Nov 14:23 rpm
lastenhuone:SailfishOS-i486-x86 sven$ ./qmake
Project ERROR: Could not connect to MerSDK Virtual Machine. No private key file
given.
lastenhuone:SailfishOS-i486-x86 sven$
```

Figure 70: Running qmake from command line.

If you run qmake manually, you will find out that it tries to connect to the *Mer build engine for cross compilation*. The error also appears if the virtual machine is up and running. In the Projects settings you can see how `qmake` is invoked if started by QtCreator.

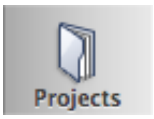


Figure 71: Project settings.

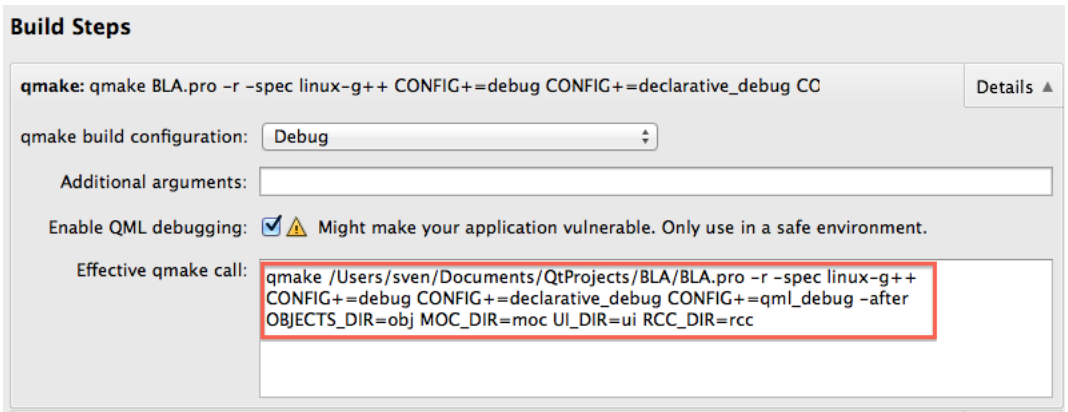


Figure 72: Build steps for qmake.

Using those parameters via command line does not work, too.

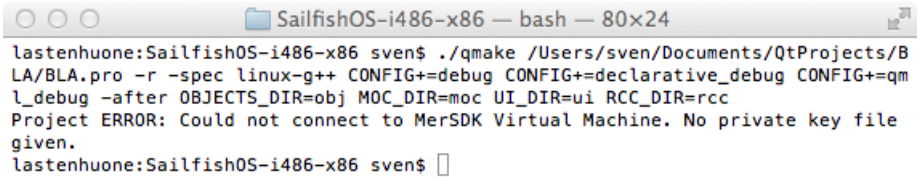


Figure 73: Running qmake from command line with parameters from build steps.

If `qmake` is invoked by the QtCreator it works just fine.

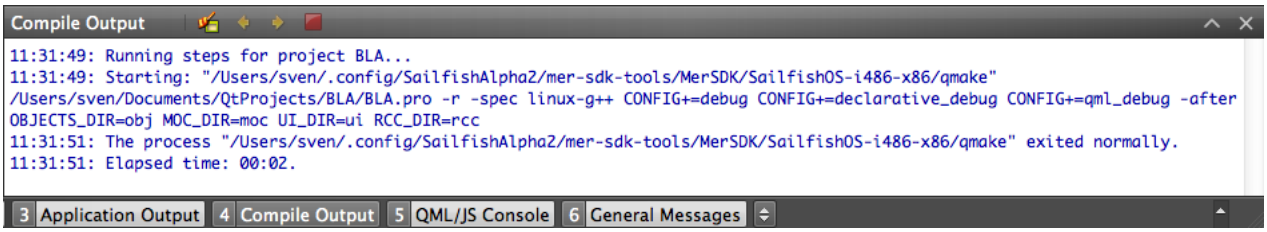


Figure 74: Running qmake from QtCreator (Build menu).

As a result you will find a `Makefile` in your project directory.

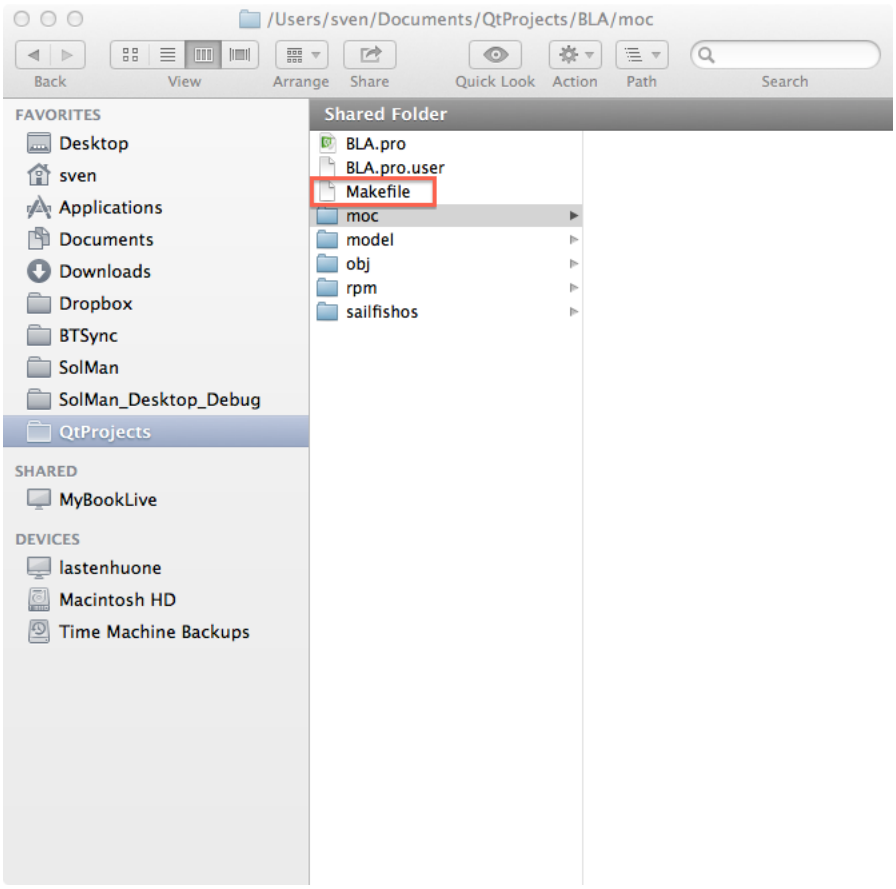


Figure 75: Result of running qmake from QtCreator (Build menu).

Here is the snippet about the MOC.

```
####### Sub-libraries

distclean: clean
	-$(DEL_FILE) $(TARGET)
	-$(DEL_FILE) Makefile


mocclean: compiler_moc_header_clean compiler_moc_source_clean

mocables: compiler_moc_header_make_all compiler_moc_source_make_all

check: first

compiler_rcc_make_all:
compiler_rcc_clean:
compiler_wayland-server-header_make_all:
compiler_wayland-server-header_clean:
```

```
compiler_wayland-client-header_make_all:
compiler_wayland-client-header_clean:
compiler_qtwayland-client-header_make_all:
compiler_qtwayland-client-header_clean:
compiler_qtwayland-server-header_make_all:
compiler_qtwayland-server-header_clean:
compiler_moc_header_make_all: moc/moc_qbusinesslogic.cpp
compiler_moc_header_clean:
	-$(DEL_FILE) moc/moc_qbusinesslogic.cpp
moc/moc_qbusinesslogic.cpp: /usr/include/qt5/QtCore/QObject \
    /usr/include/qt5/QtCore/qobject.h \
    /usr/include/qt5/QtCore/qobjectdefs.h \
    /usr/include/qt5/QtCore/qnamespace.h \
    /usr/include/qt5/QtCore/qglobal.h \
    /usr/include/qt5/QtCore/qconfig.h \
    /usr/include/qt5/QtCore/qfeatures.h \
    /usr/include/qt5/QtCore/qsystemdetection.h \
    /usr/include/qt5/QtCore/qcompilerdetection.h \
    /usr/include/qt5/QtCore/qprocessordetection.h \
    /usr/include/qt5/QtCore/qglobalstatic.h \
    /usr/include/qt5/QtCore/qatomic.h \
    /usr/include/qt5/QtCore/qbasicatomic.h \
    /usr/include/qt5/QtCore/qatomic_bootstrap.h \
    /usr/include/qt5/QtCore/qgenericatomic.h \
    /usr/include/qt5/QtCore/qatomic_msvc.h \
    /usr/include/qt5/QtCore/qatomic_integrity.h \
    /usr/include/qt5/QtCore/qoldbasicatomic.h \
    /usr/include/qt5/QtCore/qatomic_vxworks.h \
    /usr/include/qt5/QtCore/qatomic_power.h \
    /usr/include/qt5/QtCore/qatomic_alpha.h \
    /usr/include/qt5/QtCore/qatomic_armv7.h \
    /usr/include/qt5/QtCore/qatomic_armv6.h \
    /usr/include/qt5/QtCore/qatomic_armv5.h \
    /usr/include/qt5/QtCore/qatomic_bfin.h \
    /usr/include/qt5/QtCore/qatomic_ia64.h \
    /usr/include/qt5/QtCore/qatomic_mips.h \
    /usr/include/qt5/QtCore/qatomic_s390.h \
    /usr/include/qt5/QtCore/qatomic_sh4a.h \
    /usr/include/qt5/QtCore/qatomic_sparc.h \
    /usr/include/qt5/QtCore/qatomic_x86.h \
    /usr/include/qt5/QtCore/qatomic_cxx11.h \
    /usr/include/qt5/QtCore/qatomic_gcc.h \
    /usr/include/qt5/QtCore/qatomic_unix.h \
    /usr/include/qt5/QtCore/qmutex.h \
    /usr/include/qt5/QtCore/qlogging.h \
    /usr/include/qt5/QtCore/qflags.h \
    /usr/include/qt5/QtCore/qtypeinfo.h \
    /usr/include/qt5/QtCore/qtypetraits.h \
    /usr/include/qt5/QtCore/qsysinfo.h \
```

```
    /usr/include/qt5/QtCore/qobjectdefs_impl.h \
    /usr/include/qt5/QtCore/qstring.h \
    /usr/include/qt5/QtCore/qchar.h \
    /usr/include/qt5/QtCore/qbytearray.h \
    /usr/include/qt5/QtCore/qrefcount.h \
    /usr/include/qt5/QtCore/qarraydata.h \
    /usr/include/qt5/QtCore/qstringbuilder.h \
    /usr/include/qt5/QtCore/qlist.h \
    /usr/include/qt5/QtCore/qalgorithms.h \
    /usr/include/qt5/QtCore/qiterator.h \
    /usr/include/qt5/QtCore/qcoreevent.h \
    /usr/include/qt5/QtCore/qscopedpointer.h \
    /usr/include/qt5/QtCore/qmetatype.h \
    /usr/include/qt5/QtCore/qvarlengtharray.h \
    /usr/include/qt5/QtCore/qcontainerfwd.h \
    /usr/include/qt5/QtCore/qisenum.h \
    /usr/include/qt5/QtCore/qobject_impl.h \
    model/qt/qbusinesslogic.h
  /usr/lib/qt5/bin/moc $(DEFINES) $(INCPATH) -I/usr/lib/gcc/i486-
    meego-linux/4.6.4/../../../../include/c++/4.6.4 -I/usr/lib/gcc/
    i486-meego-linux/4.6.4/../../../../include/c++/4.6.4/i486-meego-
    linux -I/usr/lib/gcc/i486-meego-linux/4.6.4/../../../../include/c
    ++/4.6.4/backward -I/usr/lib/gcc/i486-meego-linux/4.6.4/include -I
    /usr/local/include -I/usr/include model/qt/qbusinesslogic.h -o moc
    /moc_qbusinesslogic.cpp
```

The `qmake` from the SailfishOS SDK is just a simple bash script, that invokes `merssh`.

```
#!/bin/bash
exec "/Users/sven/SailfishOS/bin/Qt Creator.app/Contents/MacOS/../
    Resources/merssh" -sdktoolsdir "/Users/sven/.config/SailfishAlpha3
    /mer-sdk-tools/MerSDK" -commandtype mb2 -mertarget SailfishOS-i486
    -x86 qmake $@oluhuone:SailfishOS-i486-x86
```

So that's the trick: `$@` is replaced with the `qmake` call parameters, `oluhuone` is just the name of one of my computers.

4.2.3 .pro file

Project files contain all the information required by qmake to build your application, library, or plugin. Generally, you use a series of declarations to specify the resources in the project, but support for simple programming constructs enables you to describe different build processes for different platforms and environments[QtCreator help].

If you select the help mode of QtCreator and switch to *Index*, you can search amongst other things for `qmake` and have a look at the `qmake Variable Reference`.

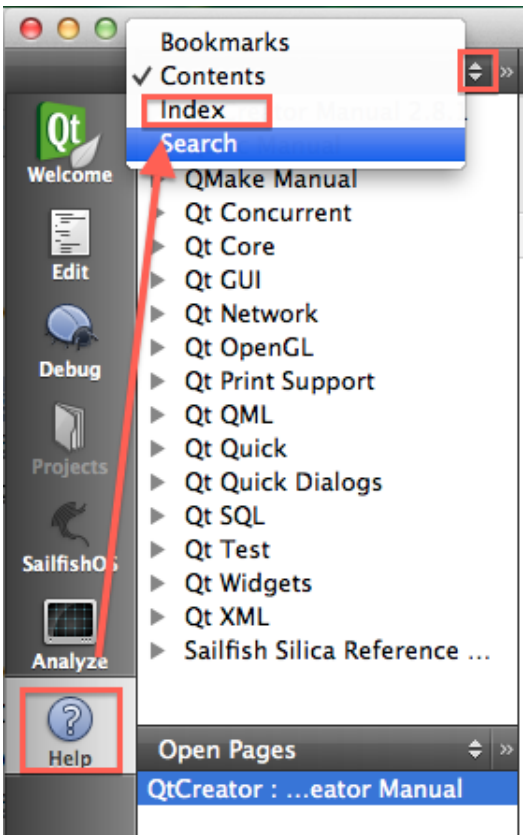


Figure 76: QtCreator, Help, Index.

The fundamental behavior of qmake is influenced by variable declarations that define the build process of each project. Some of these declare resources, such as headers and source files, that are common to each platform. Others are used to customize the behavior of compilers and linkers on specific platforms[QtCreator help].

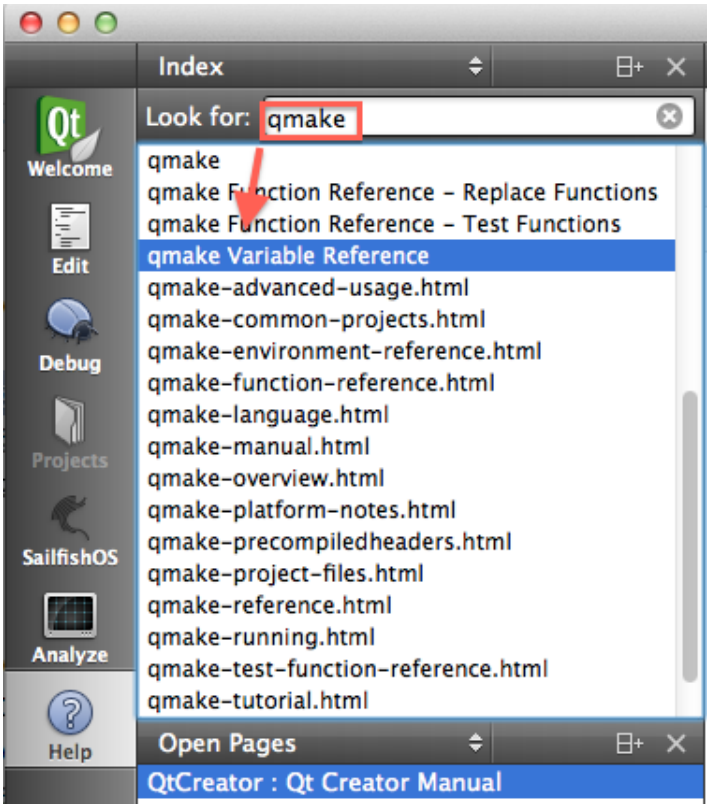


Figure 77: QtCreator, search for qmake.

The `.pro` file from a new created SailfishOS application looks like

```
# The name of your app.
# NOTICE: name defined in TARGET has a corresponding QML filename.
#         If name defined in TARGET is changed, following needs to be
#         done to match new name:
#         - corresponding QML filename must be changed
#         - desktop icon filename must be changed
#         - desktop filename must be changed
#         - icon definition filename in desktop file must be changed
TARGET = NAME_OF_THEAPP

CONFIG += sailfishapp

SOURCES += src/NAME_OF_THEAPP.cpp

OTHER_FILES += qml/NAME_OF_THEAPP.qml \
    qml/cover/CoverPage.qml \
    qml/pages/FirstPage.qml \
    qml/pages/SecondPage.qml \
    rpm/NAME_OF_THEAPP.spec \
```

```
    rpm/NAME_OF_THEAPP.yaml \
    NAME_OF_THEAPP.desktop
```

The target is the name of the compiled binary.

```
TARGET = NAME_OF_THEAPP
```

All SailfishOS silica application share some common settings, those are imported with the `sailfishapp.prf` Qt feature file.

```
CONFIG += sailfishapp
```

Your C++ implementation files are SOURCES[9].

```
SOURCES += src/NAME_OF_THEAPP.cpp
```

All kind of resource files are the `OTHER_FILES`.

```
OTHER_FILES += qml/NAME_OF_THEAPP.qml \
    qml/cover/CoverPage.qml \
    qml/pages/FirstPage.qml \
    qml/pages/SecondPage.qml \
    rpm/NAME_OF_THEAPP.spec \
    rpm/NAME_OF_THEAPP.yaml \
    NAME_OF_THEAPP.desktop
```

More directives can be found in the help system.

4.2.4 .prf file

You may ask yourself what the `CONFIG += sailfishapp` line does? It tells `qmake` to include the content of the `sailfishapp.prf` file while it processes the `.pro` file of your application. Those files are called Qt feature files and work more or less like `.pro` files. Their purpose is to share common configuration between different projects.

Have a look into the

```
~/SailfishOS/mersdk/targets/SailfishOS-i486-x86/usr/share/qt5/mkspecs
    /features/
```

[9]The corresponding header files are HEADERS.

folder, there are some more[10].

```
#
# sailfishapp.prf: Qt Feature file for libsailfishapp
# Usage: CONFIG += sailfishapp
#
# Copyright (c) 2013 Jolla Ltd.
# Contact: Thomas Perl <thomas.perl@jollamobile.com>
# All rights reserved.
#
# This file is part of libsailfishapp
#
# You may use this file under the terms of BSD license as follows:
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
   are met:
#     * Redistributions of source code must retain the above
   copyright
#       notice, this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above
   copyright
#       notice, this list of conditions and the following disclaimer
   in the
#       documentation and/or other materials provided with the
   distribution.
#     * Neither the name of the Jolla Ltd nor the
#       names of its contributors may be used to endorse or promote
   products
#       derived from this software without specific prior written
   permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
    "AS IS" AND
# ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
   THE IMPLIED
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   ARE
# DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS
    BE LIABLE FOR
# ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
   CONSEQUENTIAL DAMAGES
# (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
   SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
   CAUSED AND
# ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
```

[10]Those are actually copies to satisfy QtCreator. During compilation the same files stored on the Mer SDK virtual machine are used. Details can be found on [sailfishos6].

```
    OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
    USE OF THIS
# SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#

QT += quick qml

target.path = /usr/bin

qml.files = qml
qml.path = /usr/share/$${TARGET}

desktop.files = $${TARGET}.desktop
desktop.path = /usr/share/applications

icon.files = $${TARGET}.png
icon.path = /usr/share/icons/hicolor/86x86/apps

INSTALLS += target qml desktop icon

CONFIG += link_pkgconfig
PKGCONFIG += sailfishapp
INCLUDEPATH += /usr/include/sailfishapp

OTHER_FILES += $$files(rpm/*)
```

First of all two Qt modules are added to the project.

```
QT += quick qml
```

Your target binary is copied to `/usr/bin` when `make` is called[11].

```
target.path = /usr/bin
```

`TARGET` is a special name in a Qt project configuration file. Other group of files can be named freely as long as they don't collide with qmake variable names. All files from the folder *qml* will be installed in the directory `/usr/share/NAME_OF_THEAPP`
The `$${TARGET}` directive resolves to the content of the variable `TARGET` and that's `NAME_OF_THEAPP` in our case here.

[11] Not really, this would happen if the program would be compiled and deployed on your development machine. See at the end of this section.

```
qml.files = qml
qml.path = /usr/share/$${TARGET}
```

The file NAME_OF_THEAPP.desktop will be installed in the folder /usr/share/applications.

```
desktop.files = $${TARGET}.desktop
desktop.path = /usr/share/applications
```

Your icon file NAME_OF_THEAPP.png will be installed in /usr/share/icons/hicolor/86x86/apps.

```
icon.files = $${TARGET}.png
icon.path = /usr/share/icons/hicolor/86x86/apps
```

If you run qmake it will build a Makefile which contains the rules to install four sets of files. First the target install set which is built-in Qt projects.

```
INSTALLS += target qml desktop icon
```

Then there are the remaining three as above self defined install sets qmk, dektop and icon. When you make your application the install sets will be processed in the given order.

This tells qmake to make use of an external library that is supported by pkg-config. The name of the external library is sailfishapp.

```
CONFIG += link_pkgconfig
PKGCONFIG += sailfishapp
```

When you compile your program the defines where #included header files are looked up.

```
INCLUDEPATH += /usr/include/sailfishapp
```

All paths that are referenced in the .prf file don't make sense on your development machine or on the Mer SDK virtual machine. They will be tweaked during building time. You can watch the command in the output tab of QtCreator, the call is actually

`make install INSTALL_ROOT=/home/deploy/installroot`
So every path given in the `.prf` file is relative to `/home/deploy/installroot` on the Mer SDK virtual machine. The files stay there after you have built the application and are the source for either *Binary Copy Deployment* or *RPM Deployment*.

4.2.5 merssh

Looking with `top` showed a process called `merssh` when `qmake` was started via QtCreator. Interesting, what's that?

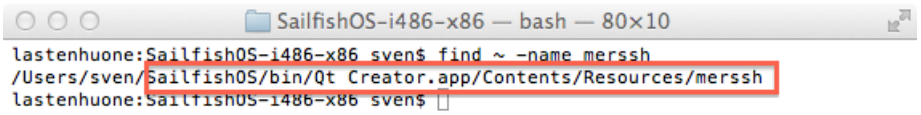


Figure 78: What is merssh?.

So it is part of the QtCreator that is shipped with the SailfishOS SDK.

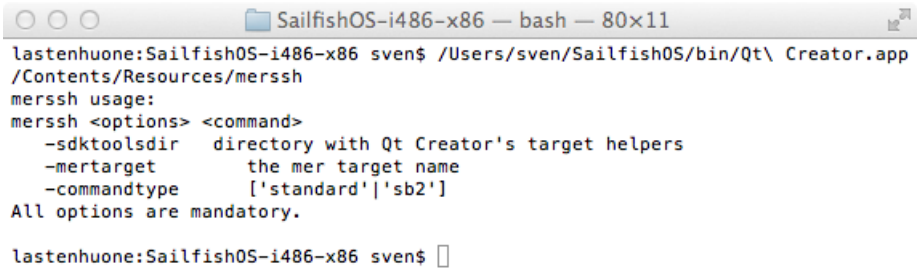


Figure 79: merssh invoked, what's it?.

All the programs called by QtCreator during the build and run process are more or less just proxy scripts[12] that call `merssh`, which in turn calls something on the Mer build engine for cross compilation, see page 76.

More than that, it calls `sb2` which is short for Scratchbox2, have a look at section Scratchbox2 on page 79, there are more details. For now let's just assume that "Scratchbox 2 is a cross-compilation engine, it can be used to create a highly flexible SDK."[sb2].

[12]OSX and Linux come with bash scripts, Windows comes with?

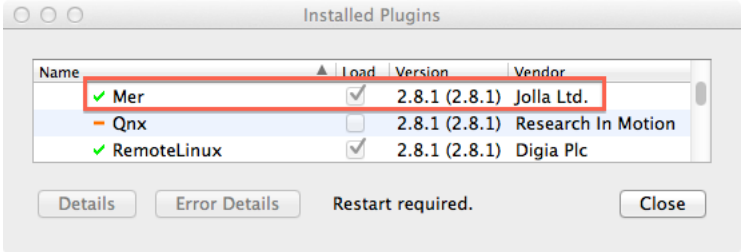


Figure 80: Mer plugin, maybe that's the source of merssh?.

I've grepped the command line for the `merssh`

```
$ ps -ef|grep "merssh"
```

And the result is:

```
/Users/sven/SailfishOS/bin/Qt Creator.app/Contents/MacOS/../Resources
   /merssh -sdktoolsdir /Users/sven/.config/SailfishAlpha3/mer-sdk-
   tools/MerSDK -commandtype mb2 -mertarget SailfishOS-i486-x86 qmake
    /Users/sven/QtProjects/TestSailfishOS/TestSailfishOS.pro -r -spec
    linux-g++ CONFIG+=debug CONFIG+=declarative_debug CONFIG+=
   qml_debug -after OBJECTS_DIR=obj MOC_DIR=moc UI_DIR=ui RCC_DIR=rcc
```

The picture is getting clearer now. QtCreator starts the SDK version of `qmake` which call `merssh` with all parameters needed to call `qmake` via `mb2` on the virtual machine.

This time it is not a bash script and I haven't found the source code yet. Maybe this little piece of software is one of the closed ones.

4.2.6 gcc

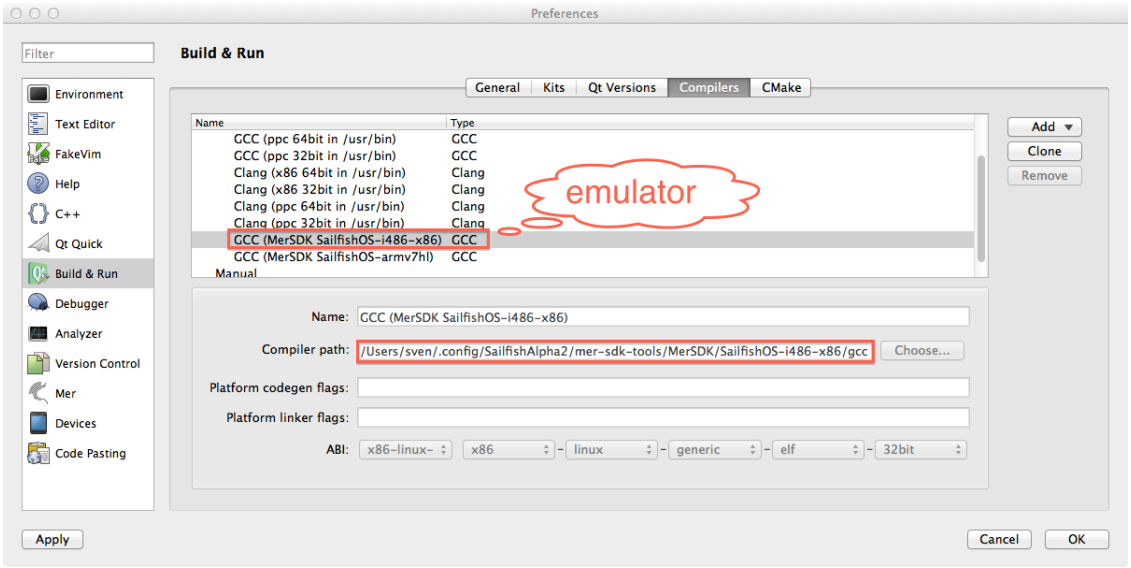


Figure 81: Preferences, compiler tab.

The SailfishOS SDK uses GCC as compiler. It is run inside the Mer build engine for cross compilation, see page 76. Stored on your development machine is only a stub or proxy that wants to connect to the virtual machine and start compiling from there.

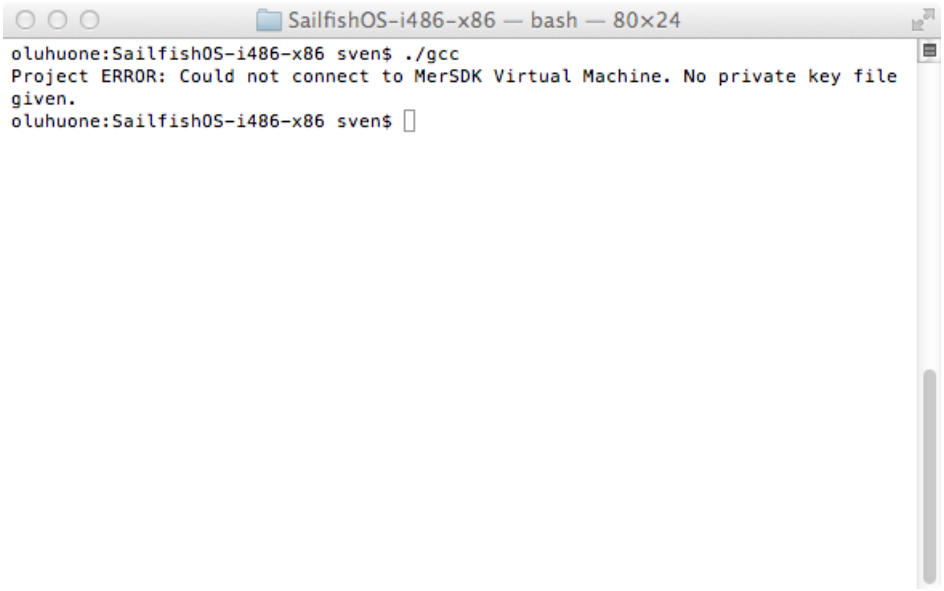


Figure 82: Running GCC from command line.

So far I don't know why this piece of software is not installed with the rest of the SDK, `~/.config` is not a directory where I would expect executables. The error message even shows up if the *Mer build engine for cross compilation* is up and running. Again this helper program is invoked via merssh, see page 59.

Looking inside `gcc` from the SDK I also find a bash script:

```
#!/bin/bash
exec "/Users/sven/SailfishOS/bin/Qt Creator.app/Contents/MacOS/../
    Resources/merssh" -sdktoolsdir "/Users/sven/.config/SailfishAlpha3
    /mer-sdk-tools/MerSDK" -commandtype sb2 -mertarget SailfishOS-i486
    -x86 gcc $@oluhuone:SailfishOS-i486-x86
```

`$@` is replaced with the `gcc` call parameters, `oluhuone` is just the name of my current machine.

One questions remains: when is this ever called? To my understanding `qmake` and `make` are called on the Mer build engine for cross compilation. I would conclude that the compiler is invoked from inside the VM.

4.2.7 make

Again, `make` is just a bash script, `$@` replaced, `oluhuone` my machine:

```
#!/bin/bash
```

```
exec "/Users/sven/SailfishOS/bin/Qt Creator.app/Contents/MacOS/../
    Resources/merssh" -sdktoolsdir "/Users/sven/.config/SailfishAlpha3
    /mer-sdk-tools/MerSDK" -commandtype mb2 -mertarget SailfishOS-i486
    -x86 make $@oluhuone:SailfishOS-i486-x86
```

I've grepped the command line for the `merssh` while building the application.

```
ps -ef|grep "merssh"
```

Resulting in

```
/Users/sven/SailfishOS/bin/Qt Creator.app/Contents/MacOS/../Resources
    /merssh -sdktoolsdir /Users/sven/.config/SailfishAlpha3/mer-sdk-
    tools/MerSDK -commandtype mb2 -mertarget SailfishOS-i486-x86 make
```

Calling `make` on the development machine just calls a proxy script, which forwards the command via `merssh` and executes `make` on the Mer build engine for cross compilation.

4.2.8 yaml / spectacle

Before the deployment process can build an RPM package of your application, it needs some information about it. This information is provided in the form of `.spec`[13] text files that are used by the `rpmbuild` program to create the package. If you would do this by hand, you would put the sources into `~/rpmbuild/SOURCES`, the `.spec` named `NAME_OF_THEAPP.spec` into `~/rpmbuild/SPECS` folder, `cd` into it and run

```
rpmbuild -ba NAME.spec
```

to build binary and source packages. Those `.spec` files are also text files but quite hard to read for humans, so in SailfishOS development you will use an intermediate format, called `.yaml`. These contain all the data needed to build a package in a more human friendly presentation. What's inside your `.yaml` depends on your `.pro` file. As a consequence, a new `.yaml` file is built every time you change your `pro` file. The base `.yaml` file of a new SailfishOS QT Quick Application project looks like

[13] RPM specifications (.spec) files.

```
Name:    NAME_OF_THEAPP
Summary: My SailfishOS Application
Version: 0.1
Release: 1
# The contents of the Group field must be one of the groups listed
   here:
# http://gitorious.org/meego-developer-tools/spectacle/blobs/master/
   data/GROUPS
Group: Qt/Qt
URL: http://example.org/
License: LICENSE
# This must be generated before uploading a package to a remote build
    service
Sources:
- '%{name}-%{version}.tar.bz2'
Description: |
  Short description of my SailfishOS Application
Configure: none
# The qtc5 builder inserts macros to allow QtCreator to have fine
# control over qmake/make execution
Builder: qtc5
# Build dependencies should ideally be specified using pkgconfig:
PkgConfigBR:
- sailfishapp >= 0.0.10
- Qt5Core
- Qt5Qml
- Qt5Quick

# Build dependencies without a pkgconfig setup can be listed here
# PkgBR:
#   - package-needed-to-build

# Runtime dependencies which are not automatically detected
Requires:
- sailfishsilica-qt5 >= 0.10.9
# All installed files
Files:
- '%{_datadir}/icons/hicolor/86x86/apps/%{name}.png'
- '%{_datadir}/applications/%{name}.desktop'
- '%{_datadir}/%{name}/qml'
- '%{_bindir}'
# This section is overwritten by Qt Creator. Do not remove the
   comments.
# Sections ends.
```

A `.yaml` text file contains information to resolve build and runtime dependencies. First we will see the directives to describe the creation of a `.spec` text file.

Required - contains the name of the application. That is not the name presented to the user, this is the name of the built binary and its relating files like icons and so on. This should *always* match the `TARGET` declaration in the `.pro` file.

```
Name:
```

Required - a short description of your application.

```
Summary:
```

Required - the version number of your package[14].

```
Version:
```

This value represents the release version of your RPM package. It is recommended to change this number each time you submit a changed package to the harbour.

```
Release:
```

Required - the group in which the application appears in the application launcher. This should always be `Qt/Qt` for a SailfishOS application.

```
Group:
```

Required - the name of the license the built package adheres to. This should be the name of the license under which you publish your application.

```
License:
```

Optional - a long(er) description of your package/application. The `|` pipe sign means that newlines in the description field shall not be converted into spaces.

```
Description:
```

Optional - the name of the tool which was used to build the package. Technically this keyword is optional but you should provide it to prevent warning messages about undefined macro files. Just leave it as it is: `qtc5`.

[14] TODO: what are the conventions? Should this always match the version in the Qt project file?

```
Builder:
```

Optional - build dependencies that make use of the assistance of `pkg-config`[fd01].

```
PkgConfigBR
```

Optional - contains the information which packages are needed at runtime. Those packages will be downloaded and installed automatically[15] when your this RMP package is installed[16].

```
Requires:
```

Optional - lists the files that are copied to the system when the RPM package is being installed. Each line listed here corresponds to one of the file groups referred to in the `INSTALL` section of your `.prf` or `.pro` file.

`%{name}` is resolved to the `Name:` entry.

```
Files:
```

4.2.9 pkg-config

```
pkg-config --list-all
```

Works just in the scratch box environment; TODO

4.2.10 rpm

Red Hat Package Manager or RPM Package Manager (RPM) is a package management system.[4] The name RPM variously refers to the .rpm file format, files in this format, software packaged in such files, and the package manager itself. RPM was intended primarily for Linux distributions; the file format is the baseline package format of the Linux Standard Base[wiki03].

Guess what, the command is a bash script inside the SDK:

[15] This is true if the package is known in the repositories of your SailfishOS device.

[16] Have a look at the Harbour rules, there are only a few packages available that you can use here.

```
#!/bin/bash
exec "/Users/sven/SailfishOS/bin/Qt Creator.app/Contents/MacOS/../
    Resources/merssh" -sdktoolsdir "/Users/sven/.config/SailfishAlpha3
    /mer-sdk-tools/MerSDK" -commandtype mb2 -mertarget SailfishOS-i486
    -x86 rpm $@oluhuone:SailfishOS-i486-x86
```

This script is called when you `deploy` your app. Deploying happens implicitly if you *run* the application. See section Run settings on page 72. Or you can deploy explicitly.

Here are some useful commands, we are examining the package `qt5-qtcore`.

```
[root@SailfishSDK ~]# rpm -q qt5-qtcore
# shows package version number
```

```
[root@SailfishSDK ~]# rpm -ql qt5-qtcore
# shows all the files contained in the package qt5-qtcore
```

```
[root@SailfishSDK ~]# rpm -qR qt5-qtcore
# shows package dependencies
```

4.2.11 deploy

There are two ways to start the deployment process, also have a look at section Run settings on page 72.

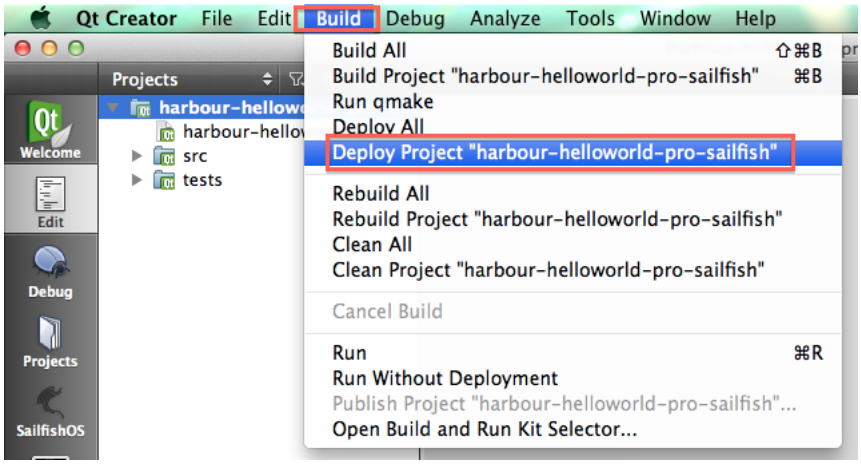


Figure 83: Deploy via “build” menu.

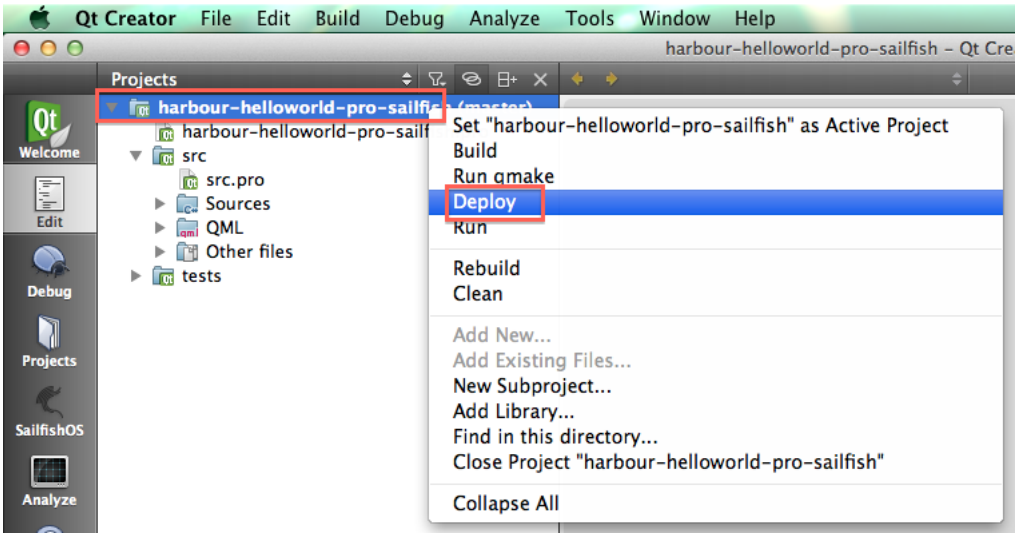


Figure 84: Deploy via “project tree”.

4.2.12 Project settings

As so often in life there is more than one way to do things. There are the 

project settings. This is the place where you define what happens when you build and compile.

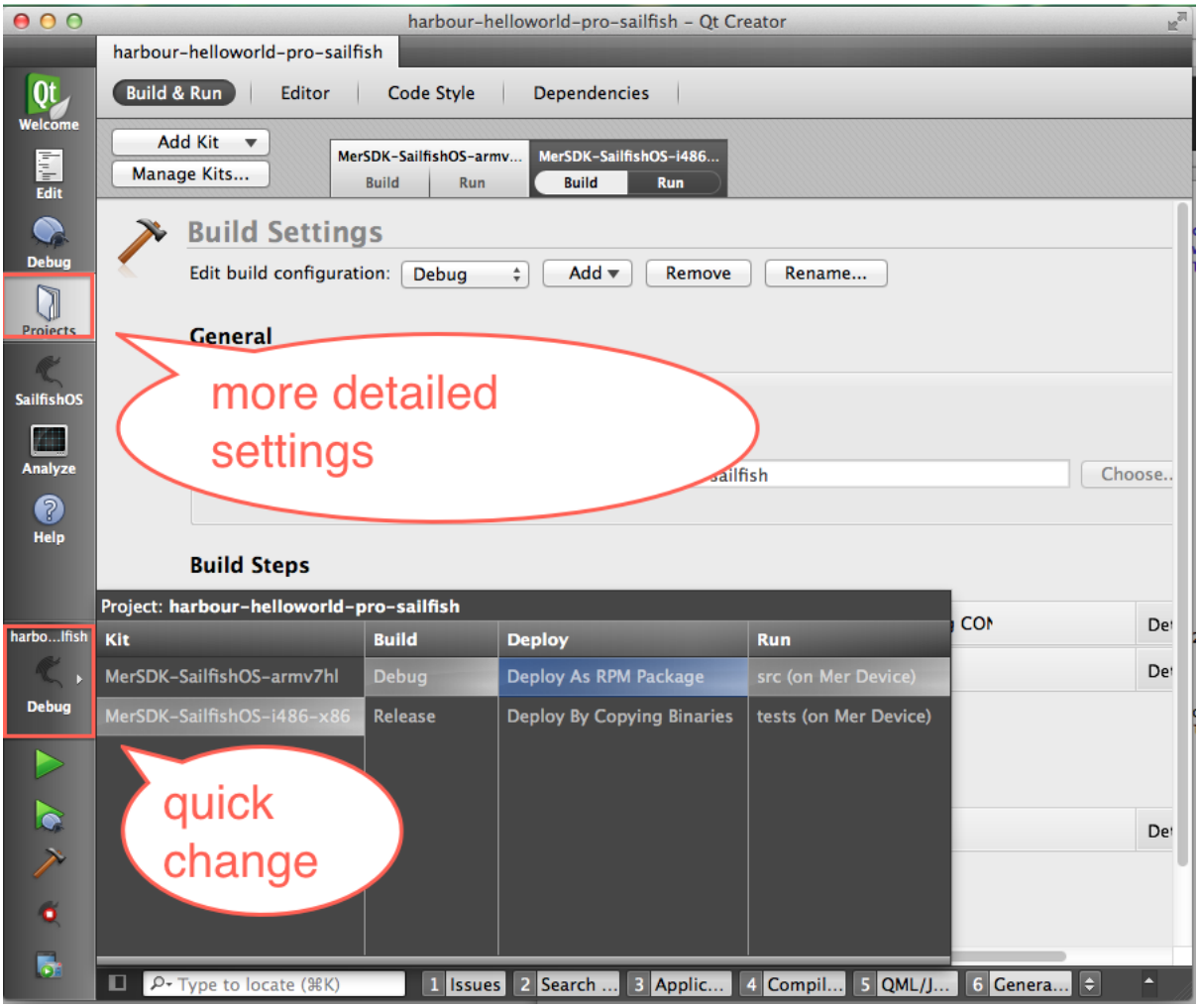


Figure 85: Two way to change project settings.

The second way is the fast *mode selector*  that switches between the settings that were defined in the *project settings*. It's the sailfish button above the green run button. You have started a project just with the `SailfishOS-i486-x86` setting[17] or meanwhile there is another platform available. In any of those cases the  button is your way to go. This way you can add new target platforms. In Qt-Speak they are called kits, a combination of Qt library, compiler and deployment target.

Right next to it is a little section for each platform you have chosen to build for. Each of this sections is decided into a *build* and *run* pane. As you can guess, *build*

[17] Or vice versa.

defines how to build your app, *run* defines how to run it.

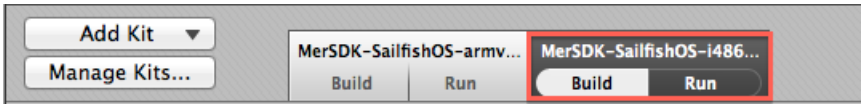


Figure 86: Settings for build and run for each target.

4.2.13 Build settings

Do you want to build a `debug` version of your app with all the debug symbol built-in? Or are you ready to ship your app to the Harbour?

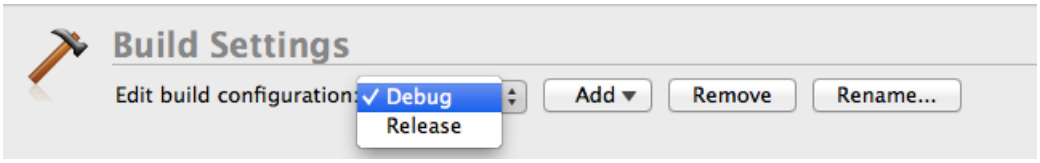


Figure 87: Build Debug or Release version?.

The build directory of your app can but must not be the same as your source directory[18]. If you check “Shadow build”, the code will be built in another directory (for each kit). This way you do not need to `make clean` after changing between the targets.

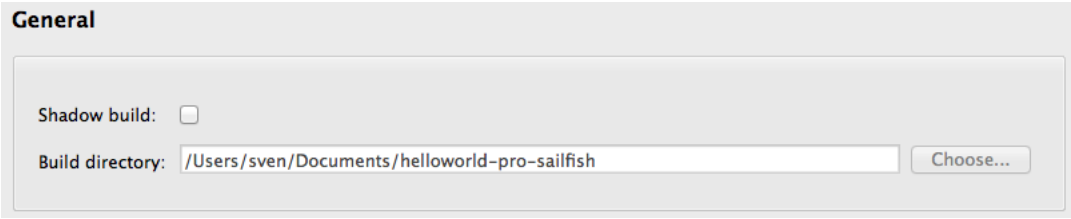


Figure 88: Where should your code be built?.

The Mer SDK Build Engine has mounted your home drive and eventually a separate source code directory from your development machine. So don’t wonder why the build directory is local on your machine?

The *Build steps* section defines what happens if you actually build your app. If you open up the details, you can see the called command line.

[18]Shadow builds work since the Alpha3 SDK.

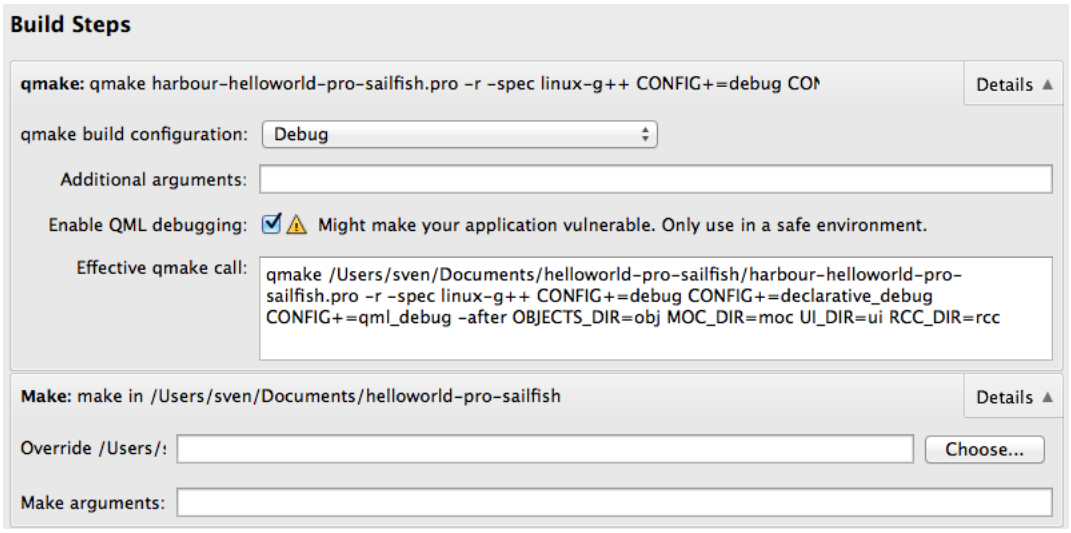


Figure 89: How to build?.

Usually qmake is called, followed by make. *Note*: those commands are not invoked locally, they run remote on the Mer build engine for cross compilation, as defined in the preferences for the QtCreator[19].

Clean steps defines how your project is `make cleaned`.

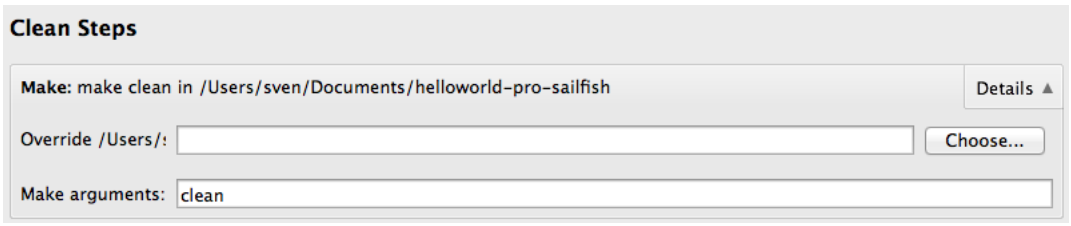


Figure 90: How to clean your project?.

The *Build environment* defines the environment variables that are used during build.

4.2.14 Pimp the clean process

Every now and then you clean your project. What bugged my for some time using QtCreator[20] was that it left the Makefile after you cleaned the project. This way `qmake` is often not run after a `make clean`. No problem, just choose the *Build*

[19]Preferences->bash scripts->merssh->VM
[20]That has nothing to do with the SailfishOS SDK, the regular QtCreator does that, too.

settings pane of the *project settings* and hit the button Add Clean Step and create an extra step that is executed every time after the cleanup has been done.

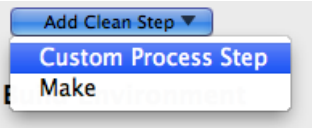


Figure 91: Create a custom process step.

Figure 92: Clean step: Remove the Makefile.

Here again for copy and paste:

```
rm
-f %{buildDir}/Makefile
%{buildDir}
```

4.2.15 Run settings

Here you have some control on how the compiled binary and its companion files will be transferred to the target device.

Choose *Deploy By Copying Binaries* if you are in an early development stage and recompile very often. It's the faster way.

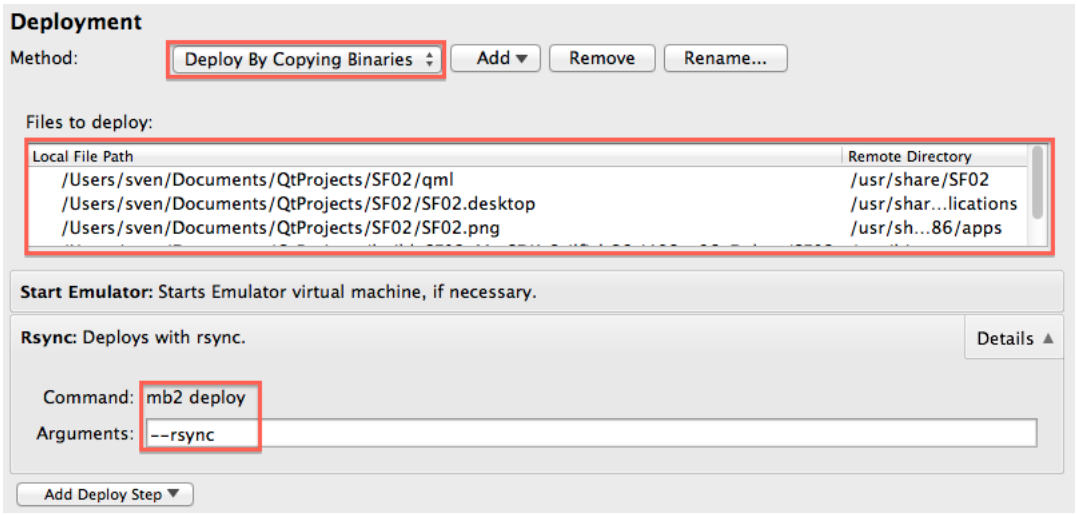


Figure 93: Run Setting, Deploy By Copying Binaries.

If your app is almost ready to ship to the Harbour, you can change to *RPM*. Now your binary and resource files will be packaged into a RPM file, transferred to the target device and installed like any other application from the store. As RPM packages can contain dependencies to other packages, you can profit from an automatic installation of those packages you depend on[21].

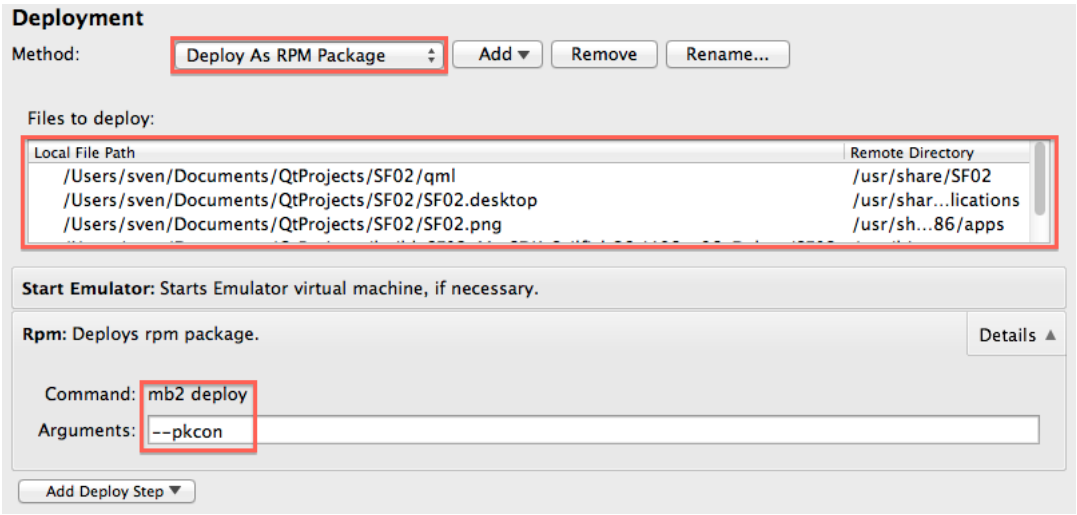


Figure 94: Run Setting, Deploy As RPM Package.

Files to deploy show which files from your computer go where on the target

[21] Have a look at section Harbour on page 96 which other packages are allowed or the other way around: which packages are not blacklisted

device. Either way uses a variation of the `mb2 deploy` command on the Mer build engine for cross compilation, which means that the binaries or packages are transferred from one VM to the other VM.

The *Run configuration* contains information about how your app is run on the target device.

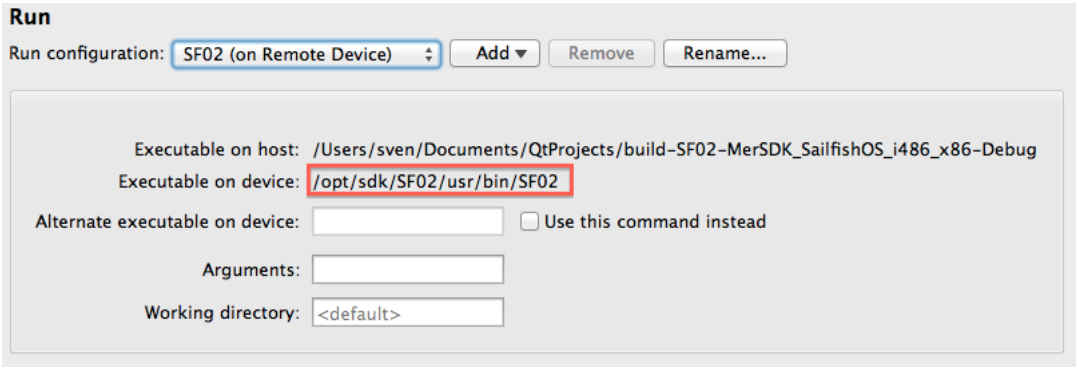


Figure 95: Run configuration.

In case your project consists of more than one (sub-)project, the *configuration* determines which of these should run. *Executable on host* is the path to the compiled binary locally on your development machine. In contrast is the *Executable on device* the path to the transferred binary on the target device[22].

The *Run environment* represents the environment variables that are visible to the binary when it is executed on the target device.

[22]If you hit *run*, the binary is copied or installed at his location.

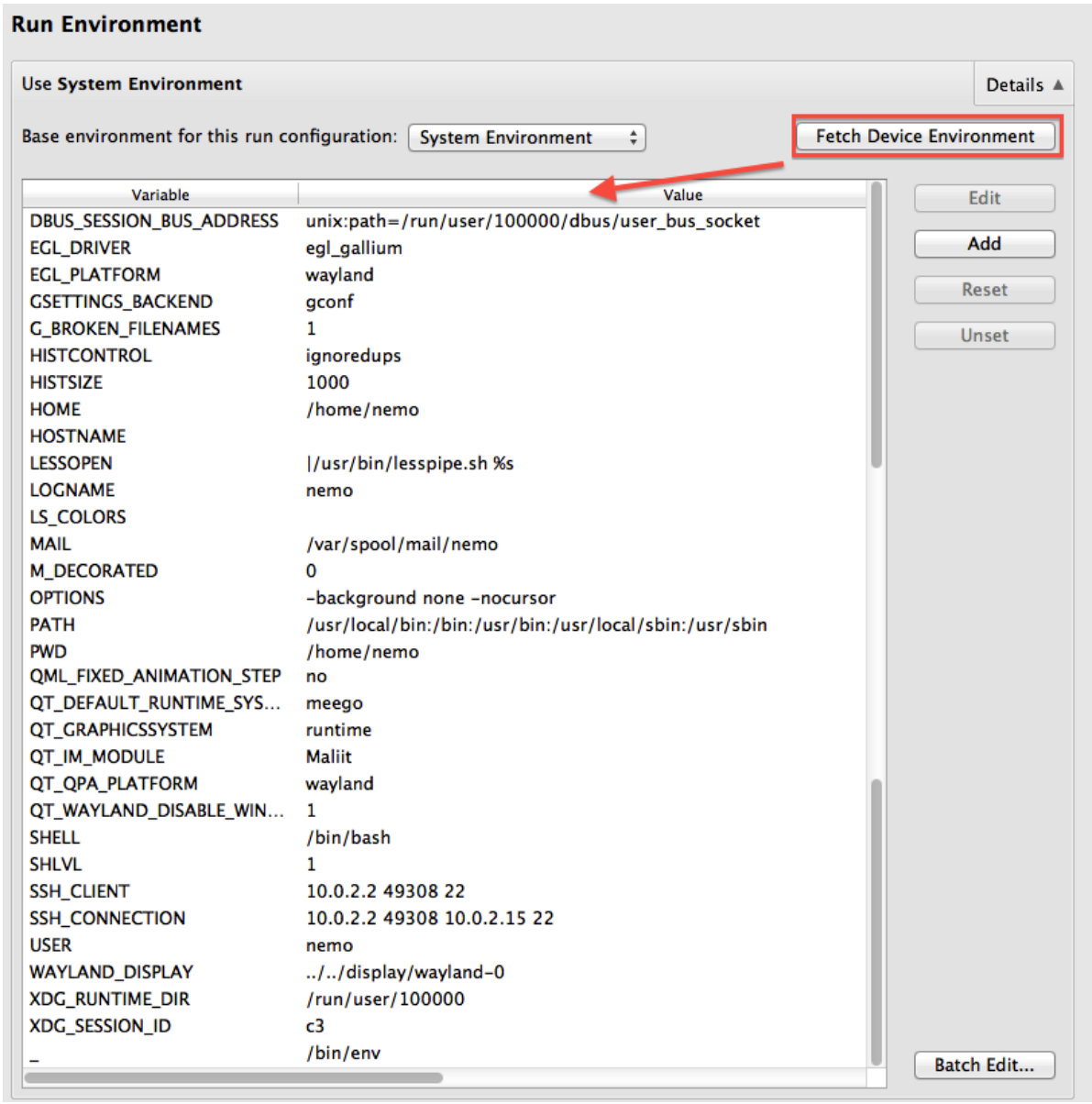


Figure 96: Run environment - environment variables on target device.

To see them, you must start the emulator and click on the Fetch Device Environment button. The *Analyzer Settings* are clearly for `valgrind` a static analyzer tool. I haven't used it one the emulator yet and on OSX Mountain Lion it does not run anymore. I'd prefer `clang` which is AFAIK not available on the SDK.

The *Debugger Settings* are for TODO (the checkboxes are pretty cleat that it's about switching debugger support on and off but I haven't used it in this environment so far - so I don't make things up here).

4.3 Mer build engine for cross compilation

"The Mer build engine is a virtual machine (VM) containing the Mer development toolchains and tools. It also includes a SailfishOS target for building and running Sailfish and QML applications. The target is mounted as a shared folder to allow QtCreator to access the compilation target. Additionally, your home directory is shared and mounted in the VM, thus giving access to your source code for compilation. The build engine also supports additional build targets and cross-compilation toolchains. These can be managed from the SDK Control Centre interface within QtCreator which allows toolchains, targets and even individual target packages to be added and removed."[sailfishos3].

Figure 97: SailfishOS icon.

The VM runs headless[23], you can not see it running. For you as a developer there is a webpage served by this VM accessible through the SailfishOS icon inside QtCreator. See figure 62 on page 41.

[23] You can change that in Preferences->Mer, uncheck "Headless" and restart the VM or start in from the VirtualBox control center of you need it just once.

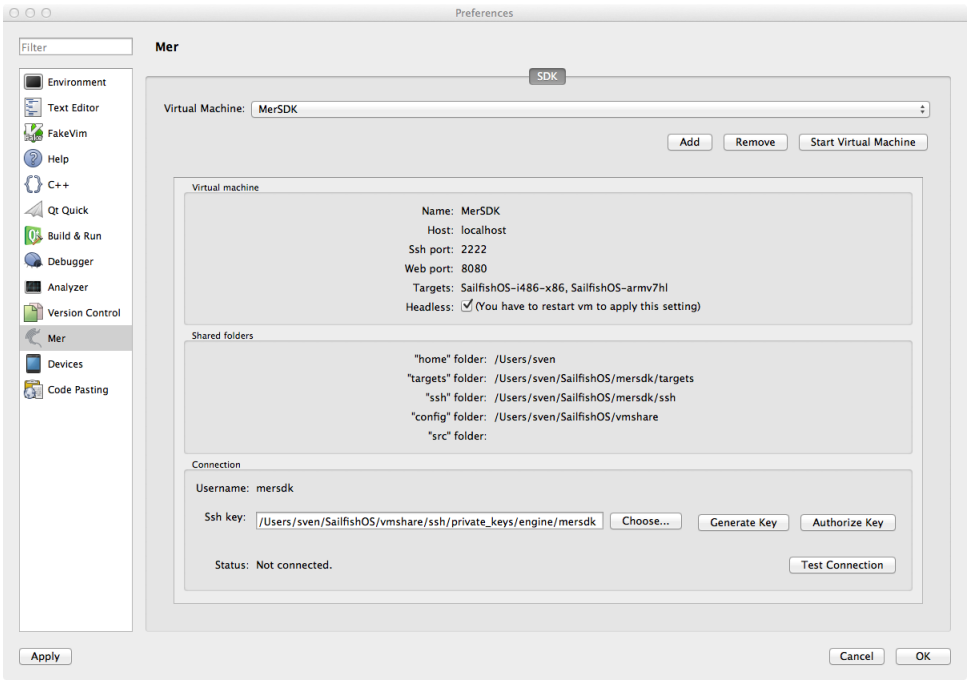


Figure 98: Preferences, Mer SDK - virtual machine.

4.3.1 Directories

VirtualBox shared folders are used for sharing files between the host and the build engine and emulators[mer04]:

- Configuration
- SSH keys
- Home directory
- Targets
- Other source directory trees

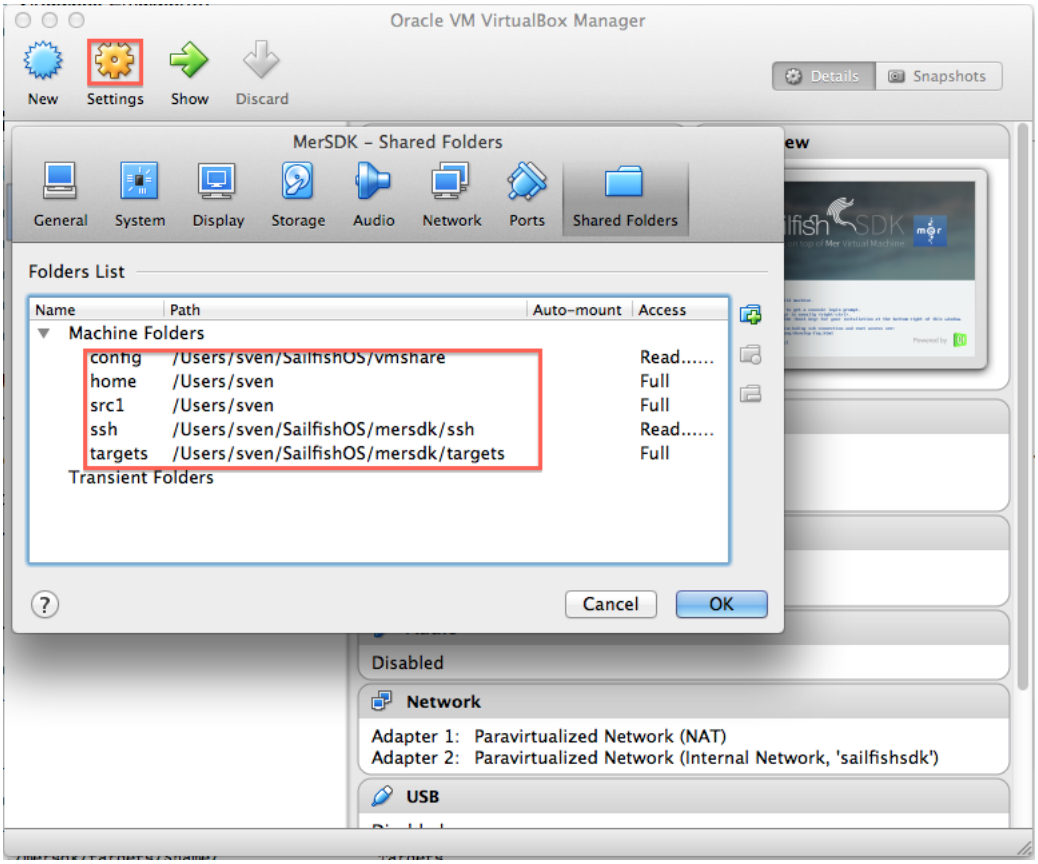


Figure 99: Virtual Box, shared folders.

Those shared folders are mount to these positions in the filesystem of the Mer build engine for cross compilation:

```
[root@SailfishSDK ~]# mount
none on /etc/ssh/authorized_keys type vboxsf (rw,nodev,relatime)
none on /home/mersdk type vboxsf (rw,nodev,relatime)
none on /host_targets type vboxsf (rw,nodev,relatime)
none on /etc/mersdk/share type vboxsf (rw,nodev,relatime)
none on /home/src1 type vboxsf (rw,nodev,relatime)
```

You can also see those and many other settings on the command line via the `VBoxManage` command.

```
VBoxManage showvminfo "MerSDK"
```

4.4 Scratchbox2

“Scratchbox2 (sbox2 or sb2) is a cross-compilation toolkit designed to make embedded Linux application development easier. It also provides a full set of tools to integrate and cross-compile an entire Linux distribution.

In the Linux world, when building software, many parameters are auto-detected based on the host system (like installed libraries and system configurations), through autotools "./configure" scripts for example. But so, when one wants to build for an embedded target (cross-compilation), most of the detected parameters are incorrect (i.e. host configuration is not the same as the embedded target configuration).

Without Scratchbox2, one has to manually set many parameters and "hack" the "configure" process to be able to generate code for the embedded target.

At the opposite, Scratchbox2 allows one to set up a "virtual" environment that will trick the autotools and executables into thinking that they are directly running on the embedded target with its configuration.

Moreover, Scratchbox2 provides a technology called CPU-transparency that goes further in that area. With CPU-transparency, executables built for the host CPU or for the target CPU could be executed directly on the host with sbox2 handling the task to CPU-emulate if needed to run a program compiled for the target CPU. So, a build process could mix the usage of program built for different CPU architectures. That is especially useful when a build process requires building the program X to be able to use it to build the program Y (Example: building a Lexer that will be used to generate code for a specific package).”[wiki02]

The Wiki page of the Mer project contains a exhaustive description how to compile a program on platform A for platform B[mer01].

4.4.1 sb2

You can find Scratchbox2 or the `sb2` in the `/usr/bin` folder of the Mer build engine for cross compilation. This shell script is not called directly. The SailfishOS SDK uses mb2 as a wrapper for convenience.

4.4.2 mb2

This is a convenience wrapper script in the `/usr/bin` folder of the Mer build engine for cross compilation.. Look at the source code in section mb2 - bash script on page 117.

Here is just the usage part to grab what is happening when this script is called[24].

[24]It should be alright to show a snippet since I included the whole source with copyright notice.

```
Executes a subset of build commands in the context of an rpmbuild.
Typically called from QtCreator to perform qmake/make phases of a
 project.
Note that any other build steps in the .spec file will also be run.

<specfile> will be looked for in the current rpm/ dir. If there is
more than one it must be provided.

CWD is used as a base dir for installroot/ and RPMS/ to allow for
shadowbuilds

mb2 is aware of spectacle and will update the spec file if there is
an obvious yaml file which is newer.

mb2 build [-d] [-j <n>] [<args>]
                    : runs rpmbuild for the given spec file in the
                      given sb2 target. Produces an rpm package.
                    : -d     enable debug build
                    : -j <n> use only 'n' CPUs to build
                    : can use -s -t -p

mb2 qmake [<args>] : runs qmake in the 'build' phase
                      Note that this also verifies target
                      build dependencies are up to date
                    : can use -s -t -p

mb2 make [<args>]   : run make in the 'build' phase
                    : can use -s -t -p

mb2 deploy --zypper|--pkcon|--rsync
                    : runs the install or rpm-creation phase and
 then
                      copies/installs the relevant files to the
 device
                    : can use -s -t -p -d

mb2 run|ssh [<args>] : runs a command (on device if --device given)
 ;
                        intended for running gdb and a gdb server
                    : can use -s -t -p -d

mb2 install [<args>] : runs the 'install' phase to install to
 $buildroot
                    : can use -s -t -p
mb2 rpm [<args>]    : runs the install & rpm-creation phases
                    : can use -s -t -p
```

```
  -t | --target       : specify the sb2 target to use
  -d | --device       : specify the device
  -p | --projectdir   : when running shadow build/deploy from another
    dir
  -s | --specfile     : if the specfile is not in rpm/*.spec and
    cannot be found using -p
```

4.5 The SailfishOS Emulator

"The emulator is an x86 VM image containing a stripped down version of the target device software. It emulates most of the functions of the target device running Sailfish operating system, such as gestures, task switching and ambience theming."[sailfishos3]. At least with the AlphaSDK3 the emulator can not simulate device rotations.

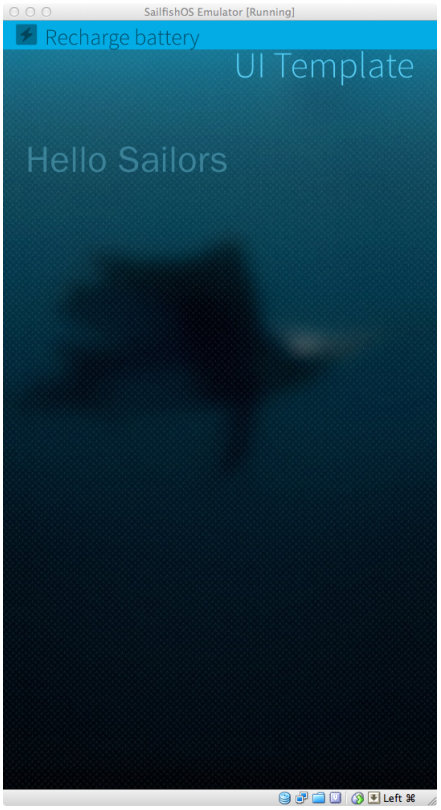


Figure 100: Emulator running the templated SailfishOS Qt Quick Application.

The Alpha3 SDK emulator brought new settings for developers.

Choose , tap on System settings and then

Finally in the developer settings you can choose a password for the user `nemo` or update the package repositories.

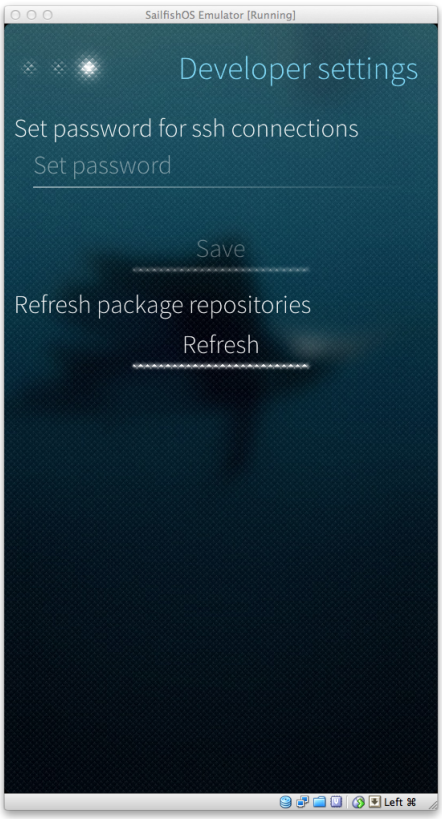


Figure 101: Choose password or update package repositories.

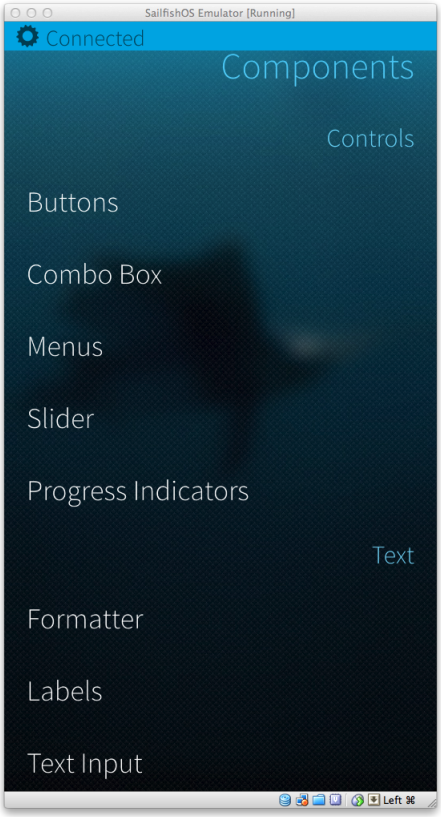
4.6 Sailfish Silica

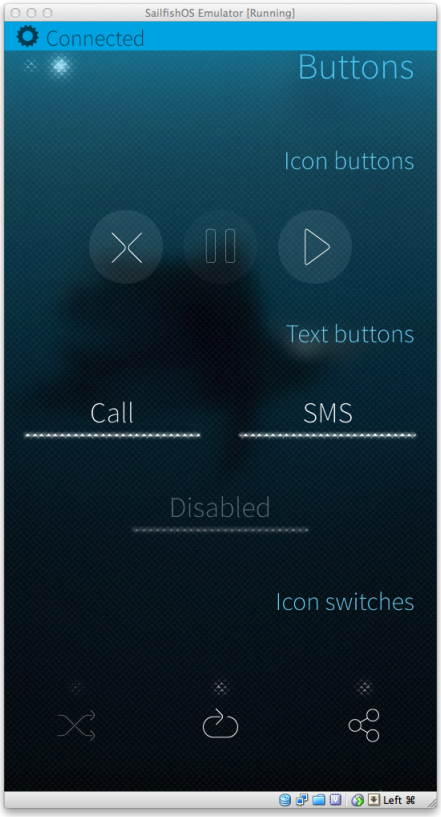
"Sailfish Silica is a QML module which provides Sailfish UI components for applications. Their look and feel fits with the Sailfish visual style and behavior and enables unique Sailfish UI application features, such as pulley menus and application covers."[sailfishos3].

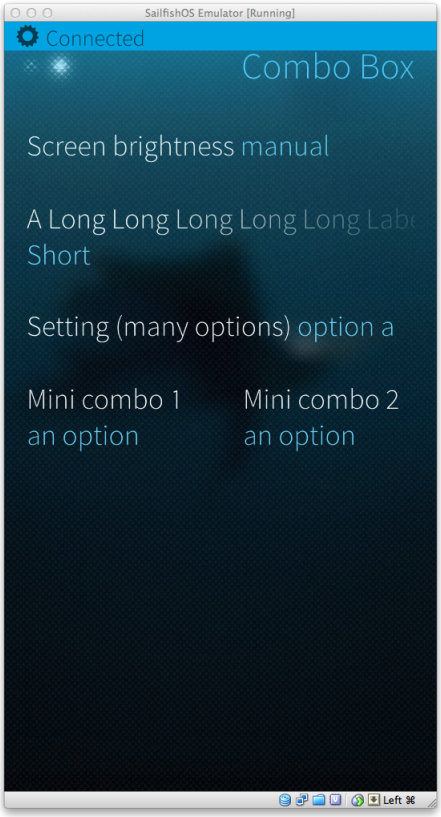
QML[qt05] is the Qt *Q*uick *M*arkup *L*anguage[qt06] that supersedes widgets for designing user interfaces. It is a declarative "language" that can contain a small subset of Javascript.

Also have a look at some open source examples on Github[sailfishos5]. The emulator comes with a demo application that shows the silica components.

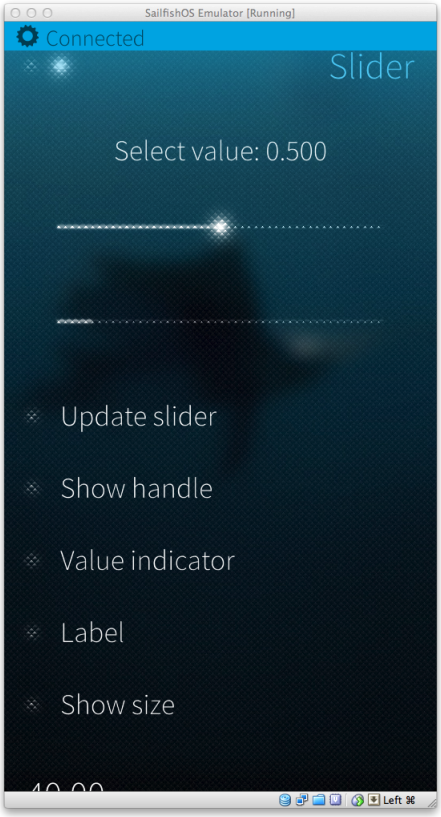

SailfishOS Emulator [Running]
Connected
Left ⌘

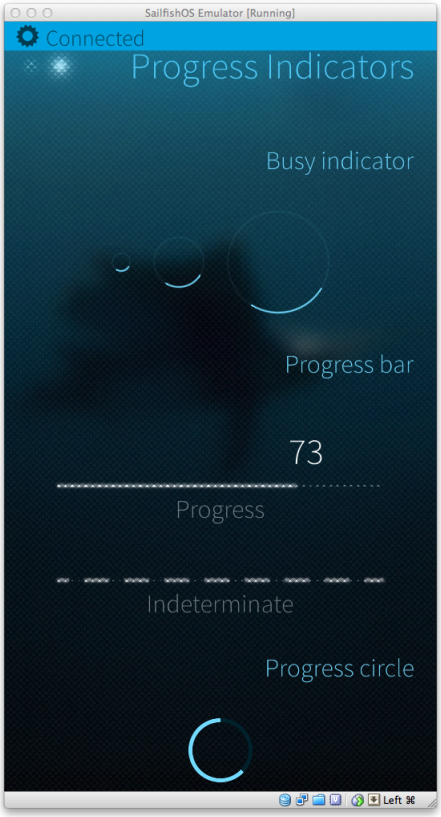

SailfishOS Emulator [Running]
Connected
Components
Controls
Buttons
Combo Box
Menus
Slider
Progress Indicators
Text
Formatter
Labels
Text Input
Left ⌘

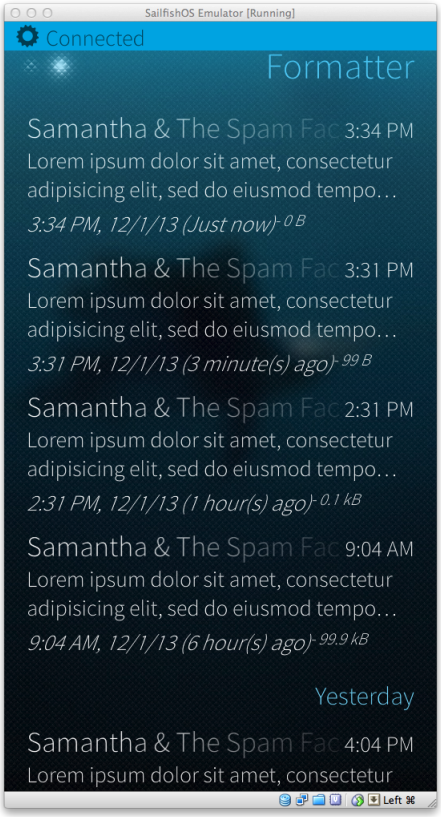

SailfishOS Emulator [Running]
Connected
Buttons
Icon buttons
Text buttons
Call
SMS
Disabled
Icon switches
Left ⌘

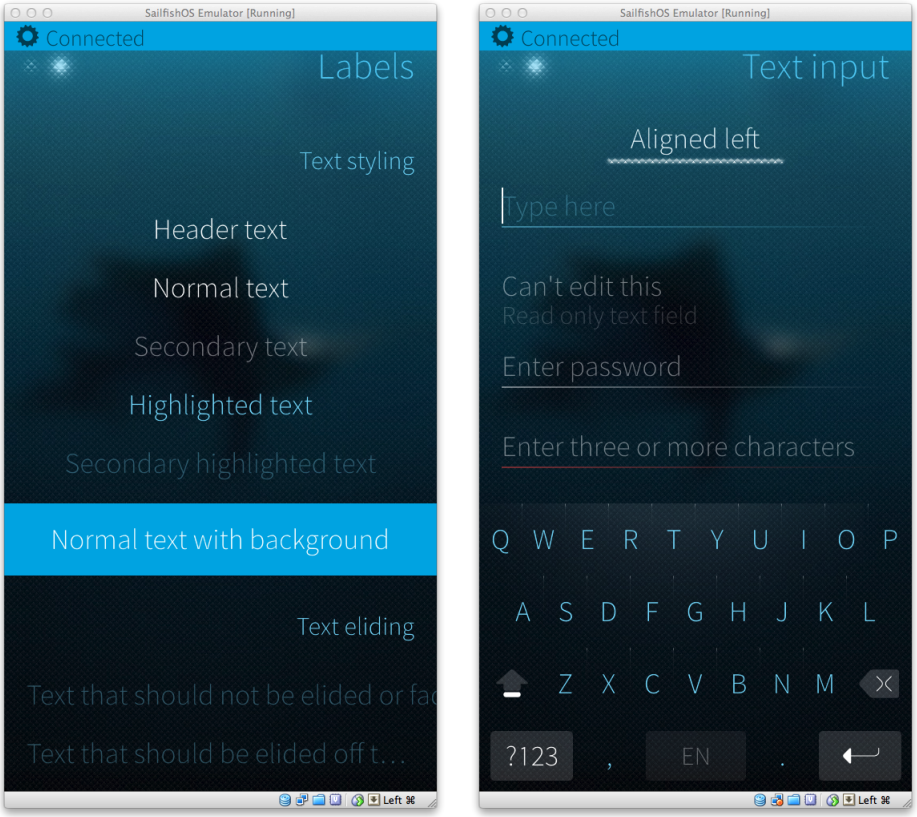

SailfishOS Emulator [Running]
Connected
Combo Box
Screen brightness manual
A Long Long Long Long Long Lab
Short
Setting (many options) option a
Mini combo 1
an option
Mini combo 2
an option
Left ⌘


SailfishOS Emulator [Running]
Connected
Menus
1. Since Long
2. Run Every
3. Planetary Civilization
4. Endangered Impacts
5. Space Every
6. Surviving Civilization
7. Obliged Become
8. Spacefaring Because
9. Exploratory Romantic


SailfishOS Emulator [Running]
Connected
Slider
Select value: 0.500
Update slider
Show handle
Value indicator
Label
Show size


SailfishOS Emulator [Running]
Connected
Progress Indicators
Busy indicator
Progress bar
73
Progress
Indeterminate
Progress circle
Left ⌘


SailfishOS Emulator [Running]
Connected
Formatter
Samantha & The Spam Fac 3:34 PM
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo…
3:34 PM, 12/1/13 (Just now) - 0 B
Samantha & The Spam Fac 3:31 PM
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo…
3:31 PM, 12/1/13 (3 minute(s) ago) - 99 B
Samantha & The Spam Fac 2:31 PM
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo…
2:31 PM, 12/1/13 (1 hour(s) ago) - 0.1 kB
Samantha & The Spam Fac 9:04 AM
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempo…
9:04 AM, 12/1/13 (6 hour(s) ago) - 99.9 kB
Yesterday
Samantha & The Spam Fac 4:04 PM
Lorem ipsum dolor sit amet, consectetur
Left ⌘



4.7 Tools chained up

Now that we have seen all the tools, bits and pieces, I will try to give an overview how everything works together, when you compile your code in QtCreator for SailfishOS.

4.7.1 Build process

As an example we just assume that the user[25] builds an app for the emulator and thus uses the `SailfishOS-i486-x86` target.

[25] That means you ;-)

QtCreator	*Mer SDK VM*	*emulator*
user starts build		
qmake		
merssh		
	mb2 -mertarget SailfishOS-i486-x86 qmake	
make		
merssh		
	mb2 -mertarget SailfishOS-i486-x86 make	
parse output		

4.7.2 Run app

Now the user hits *run*, variation 1 = *Deploy By Copying Binaries*.

QtCreator	*Mer SDK VM*	*emulator*
user runs app		
start emulator if necessary		
deploy		
merssh		
	mb2 -mertarget SailfishOS-i486-x86 deploy –rsync[26]	
	copying files to the emulator	
run executable on remote device		
		execute binary
catch execution status		

Or the user hits *run*, variation 2 = *Deploy As RPM Package*.

QtCreator	*Mer SDK VM*	*emulator*
user runs app		
start emulator if necessary		
deploy		
merssh		
	mb2 -mertarget SailfishOS-i486-x86 deploy –pkcon	
	building RPM package	
	copying RPM package to the emulator	
		installing RPM package
run executable on remote device		
		execute binary
catch execution status		

5 Installing additional packages

You can use additional libraries for your code, so you don't have to write all functionality for yourself. Some of them are available on the emulator and the physical device later on. Sadly not all library packages that are available for Nemo/Mer are usable here. Have look in section Harbour on page 96 for details about allowed packages. That does not mean, that you can not use libraries that are not part of SailfishOS. You can deliver them with your app, linked to the correct location. Mind the problems[27] that you inherit by doing so.

5.1 Emulator

You can login from your development machine via terminal and `ssh`, the emulator should be running.

```
$ ssh -p 2223 nemo@localhost
nemo@localhost's password:
# the password is nemo
,---
| SailfishOS 0.98.0.67 (i486,testing)
'---
[nemo@SailfishEmul ~]$
```

[27] Think security updates.

As an alternative you can log in the running VM of the emulator. Press CMD+F2[28][29] to change to the login screen. User and password are of course the same.

5.1.1 zypper

Additional software and libraries come in `RPM` packages and the management tool on the emulator is `zypper`. It provides "functions like repository access, dependency solving, package installation"[suse01].

```
[nemo@SailfishEmul ~]$ zypper refresh
```

will update the meta data that is stored from the package repository.

```
[nemo@SailfishEmul ~]$ zypper search
```

will show you all packages that can be installed on the emulator.

```
[nemo@SailfishEmul ~]$ zypper search boost
```

will show you all package names that contain `boost`.

```
[nemo@SailfishEmul ~]$ sudo zypper install boost-filesystem
```

will install the library `boost-filesystem` and all its dependencies on the emulator. *Note*: `boost-filesystem` is not one of the currently available libraries[30].

[28] On OSX your function keys will probably not work as regular function keys, they provide OSX functionality as printed on then, e.g. volume up/down. Go to System Preferences->Keyboard->Keyboard and check check "Use all F1, F2, etc. as standard function keys".
[29] On Windows and Linux use the CTRL Key instead of CMD.
[30] So why do I use it as an example? Honi soit qui mal y pense.

```
[nemo@SailfishEmul ~]$ sudo zypper remove boost-filesystem
```

will remove the library `boost-filesystem` and all its dependencies from the emulator. Of course you can install additional software on the emulator[31] that helps you to edit files directly or manage the filesystem better.

Note: you do not need the development packages on the emulator of physical device.

5.1.2 Known Logins

user	*password*	*comment*
nemo	nemo[32]	
root		no password needed, just works in the VM for me

5.2 Mer SDK Build Engine

Figure 102: Choose “SailfishOS” from the left pane.

5.2.1 Known Logins

user	*password*	*comment*
nemo	nemo	no private keys provided
mersdk		for use with `ssh` and private key
root		no password needed in the VM

5.2.2 SSH login

Open your terminal and enter

```
# connection as user mersdk
ssh -p 2222 -i ~/SailfishOS/vmshare/ssh/private_keys/engine/mersdk
   mersdk@localhost
-bash-3.2$
```

or

[31] As long as this software is not part of your app.

```
# connection as user root
ssh -p 2222 -i /Users/sven/SailfishOS/vmshare/ssh/private_keys/engine
    /root root@localhost
Last login: Fri Dec  6 16:41:16 2013
[root@SailfishSDK ~]#
```

5.2.3 Public keys

With the SDK come the public/private keys for SSH connections[33].

```
tree ~/SailfishOS/vmshare/ssh/private_keys
/Users/sven/SailfishOS/vmshare/ssh/private_keys
+-- SailfishOS_Emulator
|    +-- nemo
|    +-- nemo.pub
|    +-- root
|    +-- root.pub
+-- engine
     +-- mersdk
     +-- mersdk.pub
     +-- root
     +-- root.pub
```

6 QML

At the beginning I've planned to give a little introduction into QML to get you starting but recently I have discovered a much better online resource[qml01]. No need to reinvent the wheel here. Just go there and have a look, it's great!

7 Templates for QtCreator

The SailfishOS SDK comes with a template for a new SailfishOS Qt Quick Application project.

[33]There were some changes from Alpha1 to Alpha3 SDK.

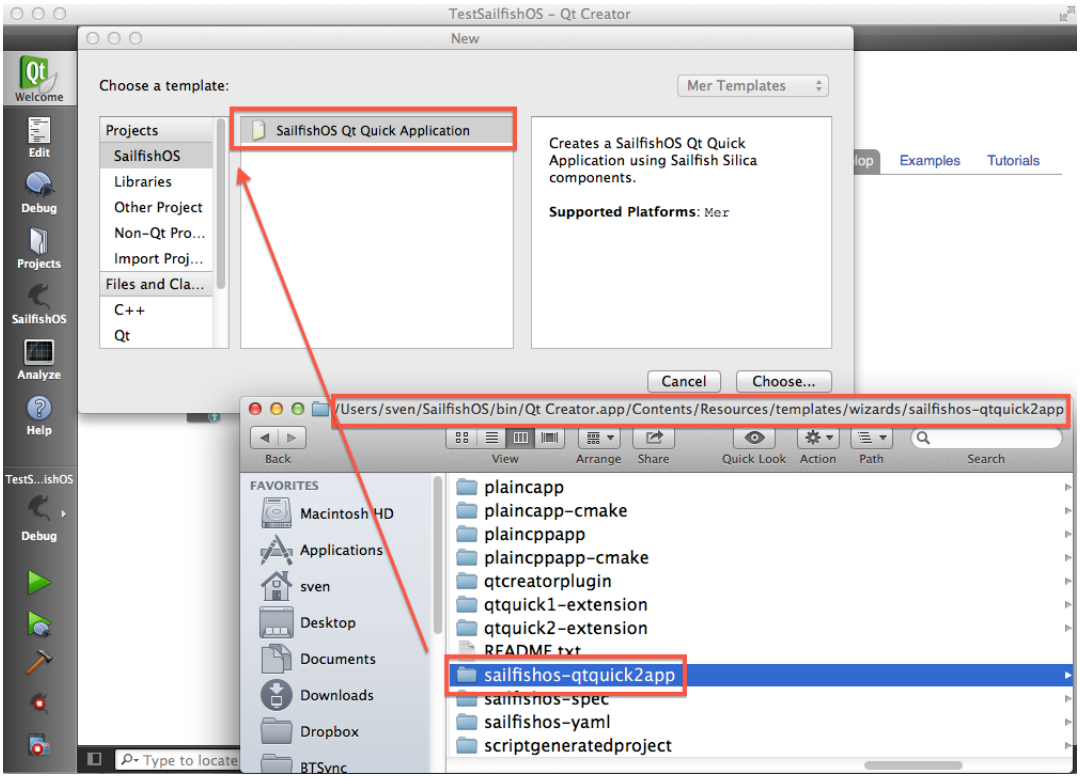


Figure 103: Template for a new SailfishOS Qt Quick Application.

7.1 OSX

On OSX those templates are stored inside the QtCreator bundle[34] on OSX. To access them you need to show the package content

[34]Bundles are just directories that end on .app are shown as one object in the finder.

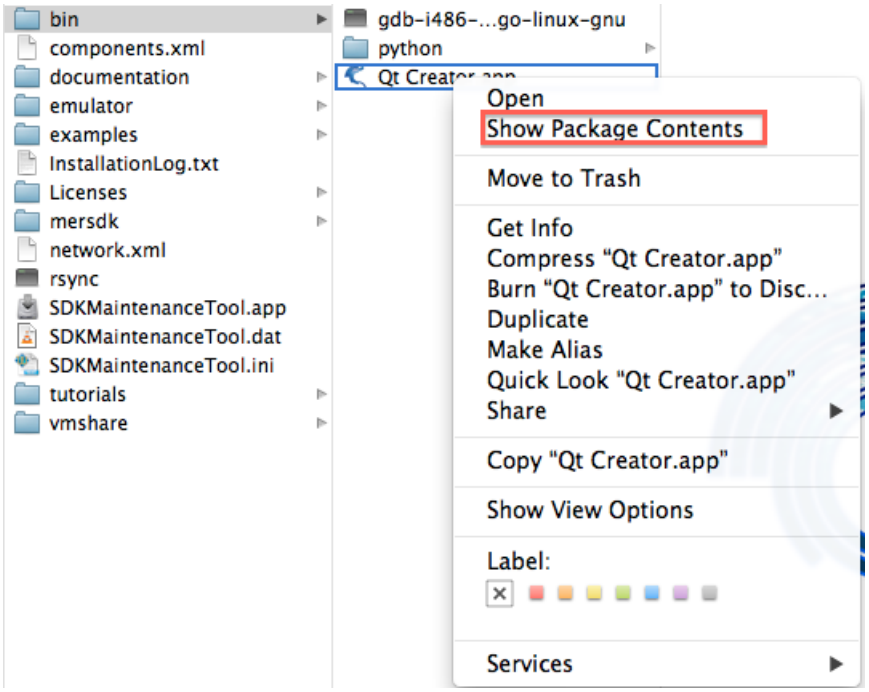


Figure 104: OSX, show package content.

or use your terminal.

```
ls ~/SailfishOS/bin/Qt\ Creator.app/Contents/Resources/templates/
   wizards
```

7.2 Linux

TODO

7.3 Windows

TODO

7.4 Generic

TODO You can change those given templates there or create new ones.

Just make sure that you don't use the names `obj`, `moc`, `ui` or `rcc` inside your template. These are going to be used to store the compile results and temporaries if you compile a SailfishOS program.

Updates of the SDK may delete changed or new templates, you might want to create a backup in a safe place and restore afterwards. There are three categories of files.

1. `wizard.xml` contains information which files end up where in your new created project, what icon is shown in QtCreator as a symbol for this project type and the information you are asked while you create the project, like e.g. project name, version number and description.

2. an icon file that represents the project type in QtCreator. That is *not* the icon of your future application. So far the template for new SailfishOS Qt Quick Application project has no specific icon.

3. template files that end up in your new created project, all `.pro`, `.h`, `.cpp`, `.qml` and `.png` files. While those files are copied, some of the information[35] can be transformed and that is also defined in the `wizard.xml` file.

TODO more details. Much more information of the format of the `wizard.xml` file can be found in [qt07].

8 Physical device

Developing with the emulator only will do you no good. You have to experience your program on a real device. Things that might look great on an emulator, may not even work on a real phone. It maybe just you fingers that are hiding the screen. Buttons are too small or too close.

8.1 How to connect to SSH over usb connection from PC

- the usb is either `usb_storage` or `usb_net`
- enable developer mode
- enable SSH (it's openssh, not dropbear)
- set password
- goto usb settings
- change that to developer mode
- reconnect usb cable

[35] Like the filename itself or details inside a file, e.g. the TARGET in the .pro file.

- you should see the ip address of the device on the UI
- you should be able to ssh to that address from PC (set an ip address first)

Taken from [ex01].

This section obviously needs a lot more information. Lacking a physical device or an SDK that enables me to interact with it, this has to be done in future.

8.2 How do I deploy a package to my phone?

As of now there is only the Alpha2 SDK out that has no built-in features to deploy a package to your device.

But you can help yourself[36]:

1. build RPM
2. deploy RPM, look in the output for the location
3. copy RPM to Jolla phone
4. use `pkcon install-local` to install it locally

Also have a look at [mer02].

9 Harbour

our wrote a fantastic app and now you want to bring it to the people. Head over to the Jolla Harbour[jolla02] and submit your app.

There are some things about naming an such to consider when you prepare your app, here is the Sailfish FAQ from the Jolla Harbour as of now = Dec. 2nd 2013. This is of course a moving target, which means there will be more libraries available in future. Even if you don't have an app ready to submit, you should look there once in a while. Ne need to re-invent the wheel here. If there is missing something, contact Jolla.

```
Naming
What should I name the application?
You have to use the prefix "harbour-" in front of the application
   name. Only lower case characters are allowed.
Why do I have to name my Application with a prefix?
The reason for this requirement is so that applications do not clash
   with other installed packages on the device. It also allows us to
   verify certain things automatically, e.g. imports of your own QML
   modules.
```

[36]Since I have no device yet, I couldn't test this. Time will come, more details will follow.

```
What is that '$NAME' you use here in the FAQ's?
That is your application name including the prefix "harbour-" (e.g.
    harbour-myawesomeapp).
How can I name the app in the application launch grid? I don't want
    that long name with prefix to appear there!
In the .desktop file, there is the "Name=" field. The string defined
    there is shown in the application launcher as the application name
    . That name does not have to be unique. So there might be more
    than one application called "HelloWorld!" in the application
    launcher grid. The package name of the application ($NAME) does
    not appear in the UI.
Where must this $NAME be used?
the executable binary: /usr/bin/$NAME
the .desktop file: /usr/share/applications/$NAME.desktop
the icon: /usr/share/icons/hicolor/86x86/apps/$NAME.png
the folder in /usr/share where you can install other application
    files: /usr/share/$NAME
The rpm package name, note that is not necessarily the same as the
    RPM file name!
The name you get with:
rpm -q --queryformat='%{NAME}\n' -p harbour-awesomeapp-1.0.0.armv7hl.
    rpm
That is what is set in .spec resp. .yaml file under "Name: "
Your own QML imports resp. modules, but with "-" replaced with "."
    due to QML grammar rules. e.g import harbour.myawesomeapp.
    MyQmlModule 1.0 as MyModule
Do I need to put that unique $NAME into the "Title" field when I
    upload an app to the Harbour?
No, in the "Title" field you can use a pretty name that will be shown
     to the user in the store client UI.
RPM-Packaging
In which locations can I install files?
You are allowed to install:
/usr/bin/$NAME <- the executable binary
/usr/share/applications/$NAME.desktop <- the desktop file
/usr/share/icons/hicolor/86x86/apps/$NAME.png <- the icon file
/usr/share/$NAME/* <- anything else (data files, private shared
    libraries, private QML imports, etc..) goes here
Why are you so restrictive? Why can't I install my libraries, images
    etc in places where I think it makes sense?
We have to ensure that rpms can be installed and do not conflict with
     other rpms. It will also allow us to install store applications
    under a different path (rpm --relocate) in the future.
What does the package name have to be? (.spec/.yaml "Name: foo")
Use "Name: $NAME". See Naming section in this FAQ.
Icons
Which size should the application icon be?
86x86. Older SDK versions contain a template which suggests a size of
     90x90. That is obsolete and will soon be updated with the next
```

```
    SDK version.
Where shall the icon be installed?
/usr/share/icons/hicolor/86x86/apps/$NAME.png. Older SDK versions
    contain a template which suggests a size of 90x90 and also a
    different install path. That is obsolete, not supported anymore,
    and will soon be updated with the next SDK version.
What file formats are supported for the icon?
The icon must be a PNG file.
How do I define an icon in the .desktop file, so it shows up in the
    application launcher?
Icon=$NAME

You must not use absolute path names. There was a bug in older SDK
    versions (lipstick < 0.18.6) so the absolute path was necessary
    and the template suggested it. That will soon be updated with the
    next SDK version. The reason to not use absolute path names is: it
     would allow us in the future to install store applications under
    a different path.

.desktop-Files
What do I have to put into the Exec= line?
Exec=$NAME (for Silica applications using C++ and QML) or Exec=
    sailfish-qml $NAME (for QML-only Silica applications without an
    application binary)
What do I have to put into the Icon= line?
Icon=$NAME
What do I have to put into the Name= line?
The name defined there is shown in the application launcher as the
    application name. That name does not have to be unique. So there
    might be more than one application called "HelloWorld!" in the
    application launcher grid.
What do I have to put into the X-Nemo-Application-Type= line and what
     is it good for?
For Silica Qt 5 applications, use "X-Nemo-Application-Type=silica-qt
    5" - this will make sure that the application is launched using
    the right booster, and will make startup faster.
How can I disable single-instance launching?
In general, you should use single-instance launching (tapping on the
    application icon will bring the existing window to the foreground)
    . If for some reason your application conflicts with single-
    instance launching, you can add "X-Nemo-Single-Instance=no" to
    your .desktop file to disable this behaviour.
QML API
Which QML modules (imports) are allowed?
Currently the following QML modules (imports) are allowed:
QtQuick 2.0
QtQuick 2.1
QtMultimedia 5.0
Sailfish.Silica 1.0
```

```
QtQuick.LocalStorage 2.0
QtQuick.XmlListModel 2.0
QtQuick.Particles 2.0
QtQuick.Window 2.0
Why do you not allow more QML modules from Qt/Nemo/Mer or other 3rd
    party?
Not all modules have a stable API. Before promising compatibility, we
     must first make sure that we can promise the API is of high
    quality and will not change (through a review process). We are
    open for suggestions to provide more APIs.
Can I use QML modules, which I ship together with the application?
Yes, you can do that. But you have to prefix the name of the imports
    with your $NAME, but with "-" replaced with "." due to QML grammar
     rules (e.g. harbour.myapp.myQmlObject, where harbour-myapp = $
    NAME). And you have to install them under /usr/share/$NAME (
    loadable QML plugins or the QML files) if the type is not built
    into the application (setContextProperty).
Shared Libraries
Which shared libraries can I link against?
Currently the following shared libraries are allowed:
libsailfishapp.so.1
libmdeclarativecache5.so.0
libQt5Quick.so.5
libQt5Qml.so.5
libQt5Network.so.5
libQt5Gui.so.5
libQt5Core.so.5
libGLESv2.so.2
libpthread.so.0
libstdc++.so.6
libm.so.6
libgcc_s.so.1
libc.so.6
ld-linux-armhf.so.3
libQt5Concurrent.so.5
libQt5Multimedia.so.5
libQt5Sql.so.5
libQt5Svg.so.5
libQt5XmlPatterns.so.5
libQt5Xml.so.5
librt.so.1
libz.so.1
libQt5DBus.so.5
Why do you allow just such a limited amount of shared libraries?
We can only whitelist libraries that have both a stable API and ABI,
    and which we are sure we can provide for the forseeable future.
    For now, the list is quite small, but as Sailfish OS matures, we
    expect to add more. We are open for suggestions to allow more
    stable shared libraries.
```

```
Can I link against shared libraries which I ship with the app
    together in the same rpm?
Yes, you can do that. But you have to install the library under /usr/
    share/$NAME/. You have to ensure yourself that the rpath in your
    executable is set correct, so the linker finds the library. Future
     versions of invoker might set the LD_LIBRARY_PATH to /usr/share/$
    NAME/, but that is not yet in place.
You do not allow QtOpenGL, what is the alternative?
QtGui in Qt 5 includes a number of classes to replace the QtOpenGL
    classes. In many cases, using the QtGui equivalents will just
    involve a renaming (QGL* -> QOpenGL*) and removing the linking
    against QtOpenGL. There are API changes involved in some cases,
    but these should not be too difficult.

Startup performance is better without using QtOpenGL, which is one
    reason we are disallowing it. This is due to its dependency on
    QtWidgets, which is quite a large library.

Why do you not allow QtWidgets?
QtWidgets is not optimized for (or well tested) on Sailfish OS.
    Furthermore, it is generally not going to result in a good user
    experience due to using non-touch-optimized controls, and software
     rendering (which will be much slower than rendering using OpenGL
    ES on Sailfish OS).
I think library XYZ would be useful for others too, I want to make
    that library available in the store. Can I submit a library only
    rpm?
No, the app store, is as the name says, an application store and not
    a shared library store. But if you think an important library is
    missing in SailfishOS and you want to see it available in the
    platform, then please join the Mer and Nemo project and make the
    library available in one of these projects. Then suggest the
    library to become a supported one (by making sure it has a stable
    API, and is well-maintained and supported).
Runtimes
Can I submit Python applications?
Currently not, there are some enablers missing for that. But we are
    working on it, to make that happen. You can support us with that
    effort, please ask in Nemo project how to help with Python.
Can I submit Perl/Ruby/$MY_FAVOURITE_LANGUAGE applications?
No, and we currently do not plan to support that. But feel free to
    request it, if there is enough interest, we might allow it in the
    future.
```

10 Bug Reports

What's the correct way to report bugs?

A good question. IMHO IRC and mailing list (see section SailfishOS on page 112) are just temporary solutions. You can also write to `developer-care@jolla.com` but right at the moment there is no developer-specific bug tracker.
There is a press release announcing a portal that should be implemented soon.
`http://jolla.com/media/documents/131219_Press_Release.pdf`

11 Troubleshooting

11.1 Mer build engine for cross compilation

11.1.1 Management Webpage

For some users (this includes me), the management webpage is not shown, instead there is an empty webpage ("about:blank"). Type "127.0.0.1:8080" as URL and hit <Enter>[37]. If it doesn't work at all (even that happened to me), then you can enter "127.0.0.1:8080" in your regular browser.

If you are in doubt that there is anything running at all, you can use `telnet` or `mmap`[nm01] to check if someone is listening on the given address and port.

```
telnet 127.0.0.1 8080
# you should see some web server welcome messages
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
```

```
nmap -p 8080 -sV localhost

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-05 18:20 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.000071s latency).
PORT     STATE SERVICE VERSION
8080/tcp open  http    WEBrick httpd 1.3.1 (Ruby 1.9.3 (2013-06-05))

Service detection performed. Please report any incorrect results at
   http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 6.49 seconds
```

Maybe your VM isn't reacting at all, see Slow on page 107.

[37] the complete URL is http://127.0.0.1:8080/C/targets/

11.1.2 SSH login pre Alpha2SDK

The first SsilfishOS SDK and the SailfishOS Alpha SDK used to store the private keys in the directory `~./ssh`. Here come some quirks that happened to me. *You should upgrade!*

You try to connect via SSH with the VM and get this:

```
ssh -p 2222 -i ~/.ssh/mer-qt-creator-rsa nemo@localhost
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle
   attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
75:51:61:e8:3a:80:41:ab:81:36:bc:45:8f:ca:56:76.
Please contact your system administrator.
Add correct host key in /Users/sven/.ssh/known_hosts to get rid of
   this message.
Offending RSA key in /Users/sven/.ssh/known_hosts:3
RSA host key for [localhost]:2222 has changed and you have requested
   strict checking.
Host key verification failed.
```

Somehow the image of the VM has changed since you last logged in. Remove it from `known_hosts` and re-connect to accept the new host key.

```
ssh-keygen -R [localhost]:2222
/Users/sven/.ssh/known_hosts updated.
Original contents retained as /Users/sven/.ssh/known_hosts.old
ssh -p 2222 -i ~/SailfishOS/vmshare/ssh/private_keys/engine/mersdk
   mersdk@localhost
The authenticity of host '[localhost]:2222 ([127.0.0.1]:2222)' can't
   be established.
RSA key fingerprint is 75:51:61:e8:3a:80:41:ab:81:36:bc:45:8f:ca
   :56:76.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[localhost]:2222' (RSA) to the list of
   known hosts.
```

It may even happen that your (local) private key is missing, check with

```
ls ~/.ssh/mer-qt-creator-rsa
ls: /Users/sven/.ssh/mer-qt-creator-rsa: No such file or directory
```

Right at the moment I am not sure when the key files were delivered since I don't have a SDK older than Alpha2 installed.

I had problems, when the public key was in the `.ssh` directory.

```
ssh -p 2222 -i ~/.ssh/mer-qt-creator-rsa root@localhost
Permission denied (publickey).
# just a generic error message, then
mv ~/.ssh/mer-qt-creator-rsa.pub ~/.ssh/mer-qt-creator-rsa.pub.backup
ssh -p 2222 -i ~/.ssh/mer-qt-creator-rsa nemo@localhost
```

The private key file may have wrong permissions, the error message is very clear about that.

```
ssh -p 2222 -i ~/.ssh/mer-qt-creator-rsa nemo@localhost
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@         WARNING: UNPROTECTED PRIVATE KEY FILE!          @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
Permissions 0644 for '/Users/sven/.ssh/mer-qt-creator-rsa' are too
    open.
It is required that your private key files are NOT accessible by
    others.
This private key will be ignored.
bad permissions: ignore key: /Users/sven/.ssh/mer-qt-creator-rsa
Permission denied (publickey).
# so change them
chmod 600 ~/.ssh/mer-qt-creator-rsa
```

11.1.3 SSH login Alpha2SDK and later

No Alpha2 specific bugs yet. If you have trouble with the private keys, please note that they are now stored in another directory[mer02].

```
~/SailfishOS/vmshare/ssh/private_keys/engine
```

11.1.4 SSH login any SDK

If your `SSH` connection still does not work, it may be that the drive with the `authorized_keys` file is not mounted. Check with `mmap`[nm01] if a `ssh` service is listening at all.

```
# this is an example of a non-working ssh daemon
nmap -p 2222 -sV localhost
```

```
Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-05 13:37 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00021s latency).
PORT     STATE SERVICE        VERSION
2222/tcp open  EtherNet/IP-1?

Service detection performed. Please report any incorrect results at
   http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 89.49 seconds
```

If you can see that the ssh daemon is not working, you can restart the VM directly from VirtualBox, log into the VM as user `root` from there, check for the daemon and start it.

```
ps -ef | grep sshd
root      533    0  2:05pm tty2    0:00.01 grep sshd
# OK, it is not running - start it
systemctl start sshd.service
# check again
ps -ef | grep sshd
root    534    1  0 2:06pm ?       0:00.01 /usr/sbin/sshd -D -f /etc/
   ssh/sshd_config_engine
root    535    1  0 2:06pm tty2    0:00.01 grep sshd
```

Now you can check from your development machine again.

```
# here the ssh daemon works
nmap -p 2222 -sV localhost

Starting Nmap 6.25 ( http://nmap.org ) at 2013-12-05 14:01 CET
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00013s latency).
PORT     STATE SERVICE VERSION
2222/tcp open  ssh     OpenSSH 5.6 (protocol 2.0)

Service detection performed. Please report any incorrect results at
   http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 0.24 seconds
```

To enable the service check if it disabled at all.

```
[root@SailfishSDK ~]# systemctl status sshd.service
sshd.service - OpenSSH Daemon
    Loaded: loaded (/lib/systemd/system/sshd.service; disabled)
```

```
#                                                    ^
#                                                    |
# look here ----------------------------------------------+
    Active: active (running) since Fri, 06 Dec 2013 16:46:45 +0000;
    28min ago
   Process: 523 ExecStartPre=/bin/bash -c if [ \( ! -s /etc/ssh/
    ssh_host_dsa_key \) -a \( ! -s /etc/ssh/ssh_host_dsa_key.pub \) -a
     \( ! -s /etc/ssh/ssh_host_rsa_key \) -a \( ! -s /etc/ssh/
    ssh_host_rsa_key.pub \) ]; then /usr/sbin/sshd-hostkeys; fi (code=
    exited, status=0/SUCCESS)
  Main PID: 527 (sshd)
    CGroup: name=systemd:/system/sshd.service
      + 527 /usr/sbin/sshd -D -f /etc/ssh/sshd_config_engine

Dec 06 16:46:45 SailfishSDK sshd[527]: Server listening on 0.0.0.0
    port 22.
Dec 06 16:46:45 SailfishSDK sshd[527]: Server listening on :: port
    22.
Dec 06 16:49:59 SailfishSDK sshd[1288]: Accepted publickey for mersdk
     from 10.0.2.2 port 49275 ssh2
Dec 06 17:11:35 SailfishSDK sshd[1315]: Connection closed by 10.0.2.2
Dec 06 17:13:09 SailfishSDK sshd[1316]: Accepted publickey for root
    from 10.0.2.2 port 49300 ssh2
[root@SailfishSDK ~]#
```

Enable the service AKA unit.

```
[root@SailfishSDK ~]# systemctl enable sshd.service
ln -s '/lib/systemd/system/sshd.service' '/etc/systemd/system/multi-
    user.target.wants/sshd.service'
# check with reboot
[root@SailfishSDK ~]# shutdown -r now
```

11.1.5 Drive(s) not mounted

Log into the VM and have a look if the drive with the `authorized_keys` file is mounted.

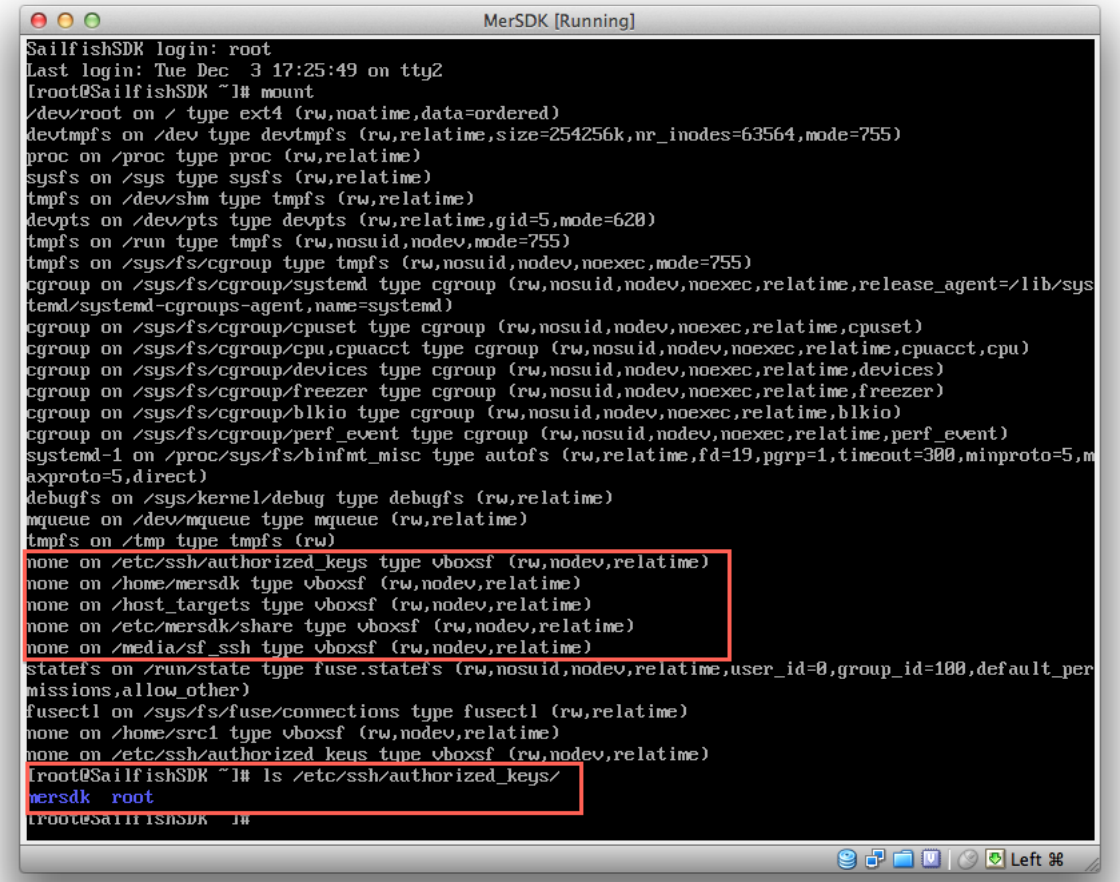


Figure 105: Check for authorized_keys.

If the drive exists[38], you can check if the file is accessible.

```
ls -lah /etc/ssh/authorized_keys/mersdk/authorized_keys
ls -lah /etc/ssh/authorized_keys/root/authorized_keys
```

Handle the other shared folders according to this. Your VM acts strange and you don't know why? Check if updates for VirtualBox are available. If yes, download and install the latest version and don't forget the extension pack.

[38]It seems that the output of mount does not show it right, even if it is mounted and accessible.

Figure 106: Check if VirtualBox update is available.

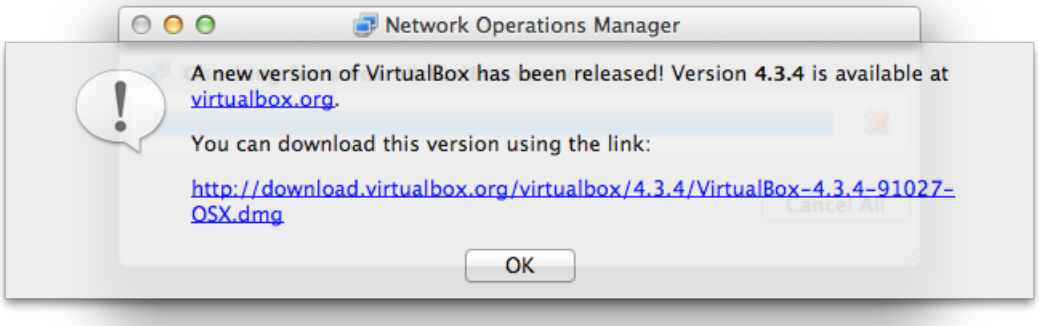


Figure 107: Update for VirtualBox is available.

11.1.6 Slow

I had the situation where everything seems perfect and still it didn't work. For me the last resort was re-installing the VM with the Mer SDK, see section Uninstall on page 113 but even that did not help in the end. One one machine the VM was running but slow[39], so every connection attempt timed out. Even control from VirtualBox itself was almost impossible.

[39] As in tar pit.

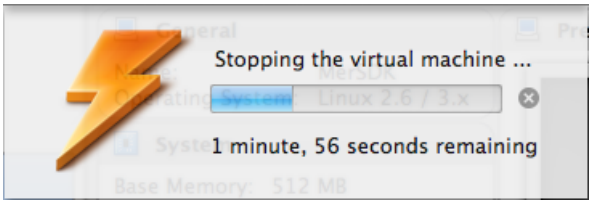


Figure 108: VM can not be controlled.

This could even happen to a working and running VM to which I successfully logged in via ssh. Working inside the terminal: no problem: compiling: no go. Until now I found no solution for this.

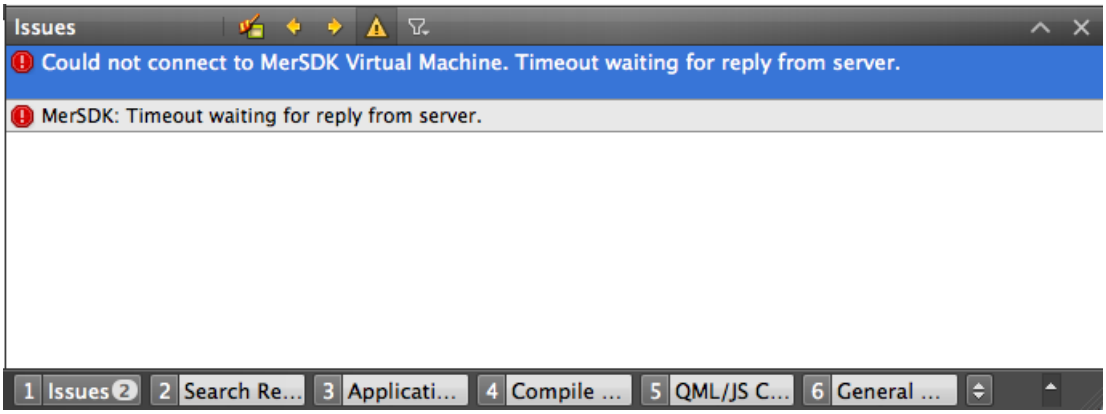


Figure 109: Project ERROR: Could not connect to MerSDK Virtual Machine. Timeout waiting for reply from server.

This happened on a notebook where the hard drive is clearly a bottleneck but it should nevertheless be fast enough to handle this task[40].

If nothing else works, power off the VM from inside VirtualBox. Even that may freeze. When the VM is in state *aborted*, kill VirtualBox and start it again. Now start the VM manually from inside VirtualBox. QtCreator will not recognize such a VM as running. Hit the  button and after its green again, your good to go.

Funny enough after looking into the logfile[41] of the VM, I found this message:

```
AIOMgr: Host limits number of active IO requests to 16. Expect a
    performance impact.
```

[40] No SSD, spinning disks with 5400 RPM.
[41] Have look in the /SailfishOS/mersdk/MerSDK/Logs folder

After some google'ing I found a hint mentioning the *I/O Host Cache* setting being off[42]. So I checked to box for a test.

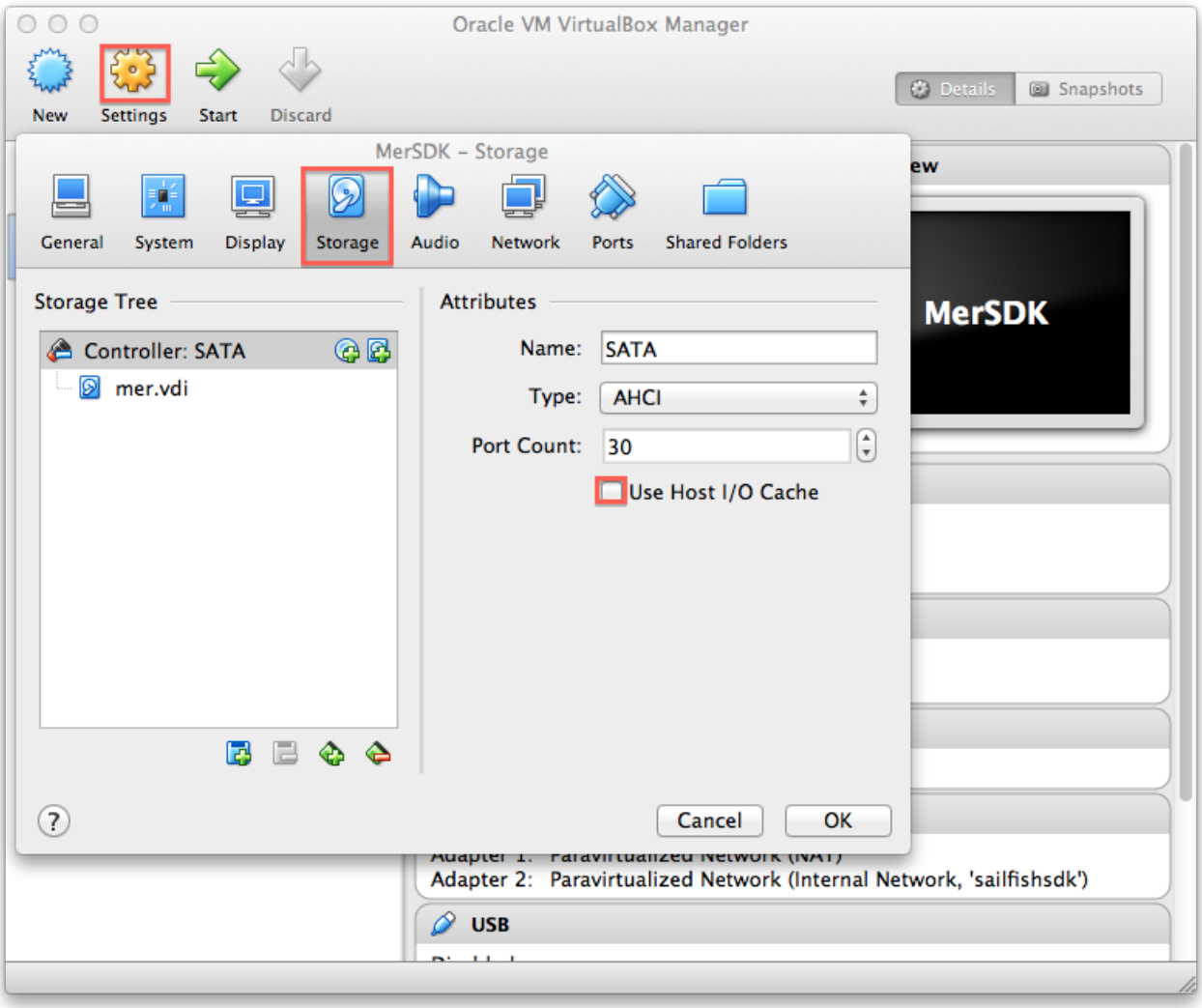


Figure 110: Turn on the host I/O cache.

Inside the VirtualBox manual is a section that may describe the cause for these problems[vbox02], as it talks about slow disk systems and that is truly the case here[43], but I must admit that I did not experience the mentioned IDE/SATA errors.

[42]This was for Vagrant which also utilizes VitualBox.
[43]From the slow notebook hard drive perspective

I am testing this right now, don't know yet if that really helps = call this experimental.

Open your terminal and run the following commands

```
VBoxManage showvminfo "MerSDK"|grep SATA
Storage Controller Name (0):            SATA
SATA (1, 0): /Users/sven/SailfishOS/mersdk/mer.vdi (UUID: fd25a075-
    e327-46b6-8657-3ab4c4294f94)
# the 1 in 'SATA (1, 0)' is what we are looking for
# it translates to the 1 in 'LUN#1' in the following command
#                                 |
#                                 +---------------------------------+
#                                                                   |
VBoxManage setextradata "MerSDK" "VBoxInternal/Devices/ahci/0/LUN#1/
    Config/FlushInterval" 5000000
# reasonable values for FlushInterval are between 1000000 (write
    often) and 10000000 (write bigger chunks)
# so I chose the middle ground (meanwhile I am at 1000000)
# if it fails after all and your VM refuses to start, you can remove
    the value with
VBoxManage setextradata "MerSDK" "VBoxInternal/Devices/ahci/0/LUN#1/
    Config/FlushInterval"
VBoxManage setextradata "MerSDK" "VBoxInternal/Devices/ahci/0/LUN#1/
    Config/IgnoreFlush"
```

This didn't solve all my problems so far but the VM got more responsive[44].

So far all tweaks have the same effect: they work after I implemented them bot only for a certain time. Then everything goes back to normal = poor performance or not working at all. But all the problems occur on 1 of 3 OSC machines I've tested the SDK on.

So this could be a some OSX machines problem only.

Another variation of this problem: MerSDK VM starts, I can login via SSH and everything is fine. After starting to build, the SSH session freezes. No CPU spikes, working directly inside the VM is still possible.

11.2 Emulator

11.2.1 Deploying offline

When you deploy an app to the emulator[45] and the deployment process discovers that you have dependent packages, it tries to download them automatically. That

[44] This comes at a price of course, the more often the VM writes the more often it will stress the host machine = your development computer.

[45] Deploy happens when you hit *run*.

works not so well if you're offline. The following lines could show up in the *Compile Output* tab.

```
Status:   Downloading packages
Package:  qt5-qttest-5.1.0+git17-1.13.1.i486
Results:
Fatal error: Download (curl) error for 'http://releases.sailfishos.
   org/sdk/latest/jolla/i486/qt/i486/qt5-qttest-5.1.0+git17-1.13.1.i
   486.rpm':
Error code: Connection failed
Error message: Couldn't resolve host 'releases.sailfishos.org'
```

At least for the first compilation of your app you may need a connection to the internet.

11.2.2 Timeout, emulator offline

The emulator is running as you can see but QtCreator complains that it is not reachable. That is a well known bug, just hit the start button in QtCreator and try again.

12 Use your editor of choice

Now you are happy with coding for a device with SailfishOS and the SDK, API and all, but for some reason you want to use *your* editor[46] of choice. Because you are used to it, never used something else, it's better, whatever.

Well, I don't know what editor you are using and providing instructions for all them seems unfeasible. Instead I will try to walk you through all the build/compile/steps on the command line. This way you can see what's really happening behind the scenes and can integrate those steps into the environment of your choice. There are at least two ways to do this. First is the way QtCreator goes: call wrapper scripts. The second is to really walk barefooted.

12.0.3 QtCreator way

TODO

12.0.4 Going barefooted

TODO

[46] notepadl vi, emacs, eclipse, named it.

13 SDK / API (TODO)

13.0.5 Application lifecycle

TODO

13.0.6 Gyroscope - device rotation

TODO

13.0.7 GPS - where am I?

TODO

13.0.8 Bluetooth

TODO

13.0.9 NFC

TODO

13.0.10 The other half

TODO

14 Community

Much of the communication is done via `IRC`[irc01], you should make yourself familiar with that and follow the guidelines of the respective channels. You will see that not each of them is as chatty as a marketplace.

14.1 Jolla

Jolla homepage: `http://jolla.com`.
Jolla on Twitter: `https://twitter.com/JollaHQ`.

14.1.1 SailfishOS

As a developer you should subscribe to the mailing list at `https://lists.sailfishos.org/cgi-bin/mailman/listinfo/devel`.

Have a look at the Wiki at `https://sailfishos.org/wiki/Main_Page`.
At freenode: `#sailfishos`.

If those don't match, there is still `developer-care@jolla.com`.

14.1.2 Mer

At freenode: `#mer`. Homepage: `http://merproject.org`.

14.1.3 Nemo mobile

At freenode: `#nemomobile`. Mer Wiki page about the Nemo project: `https://wiki.merproject.org/wiki/Nemo`.

15 Uninstall

You really want to quit? Stay here, the water is warm! Jokes aside, for whatever reason you might want to uninstall the SDK, here is how is goes.

15.0.4 OSX

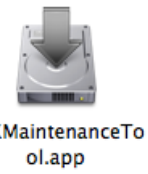


Figure 111: Inside the SailfishOS folder you find the maintenance application. Start it to uninstall the SDK.

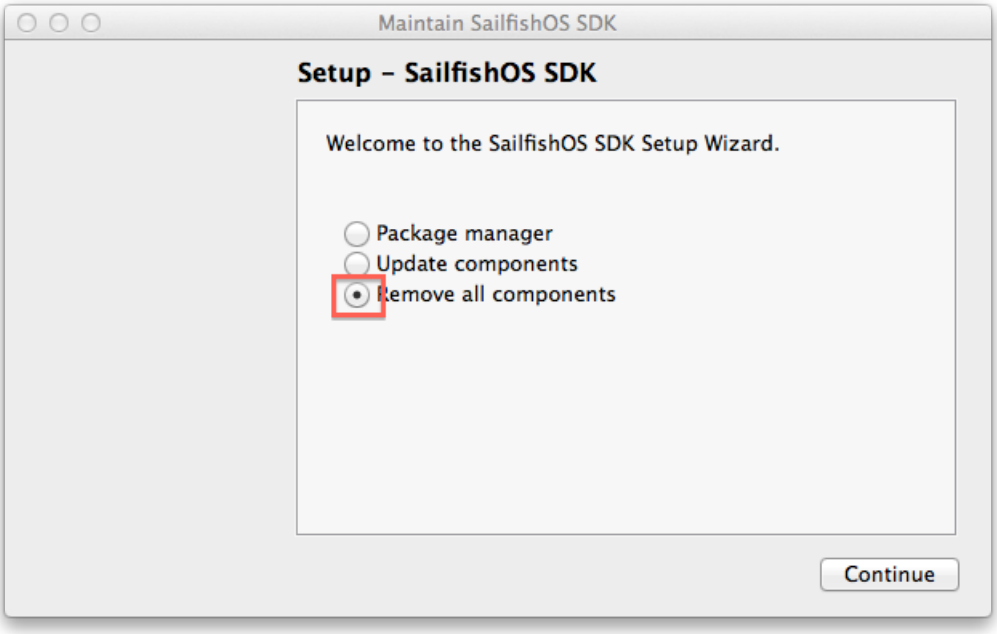


Figure 112: Uninstall, step 1.

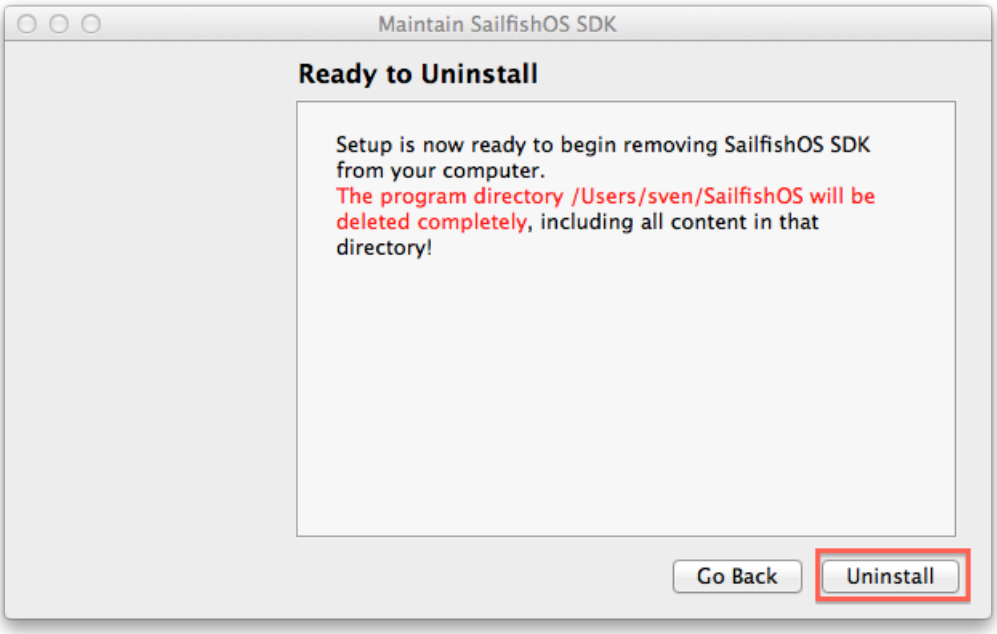


Figure 113: Uninstall, step 2.

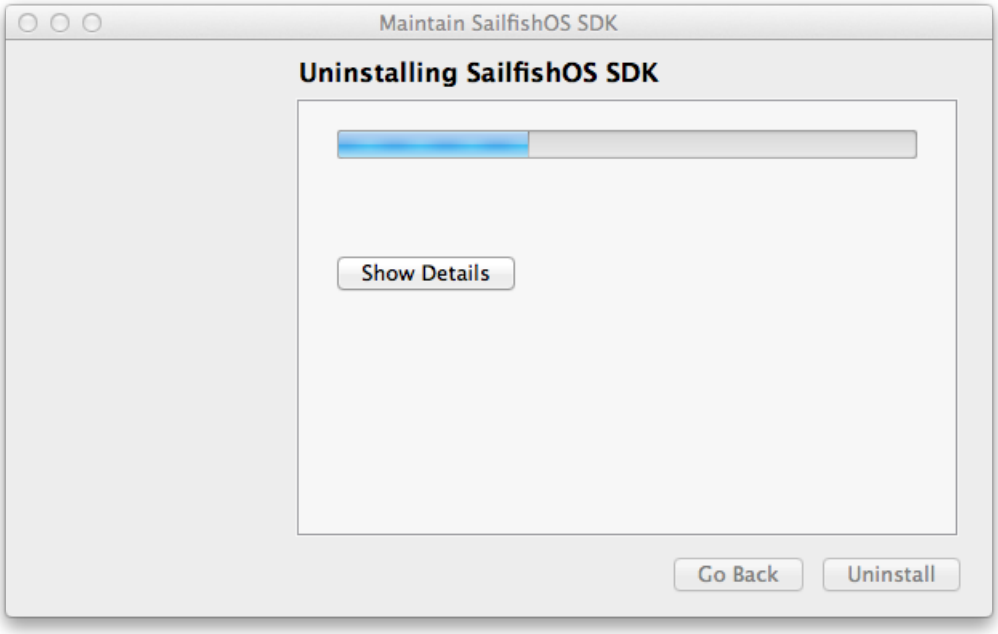


Figure 114: Uninstall, step 3.

Figure 115: Uninstall, step 4.

To make it really clean, remove orphaned files and directories. Of course you can keep those files for later use.

```
# remains from the pre Alpha2 SDK
rm ~/.ssh/mer-qt-creator-rsa*
# there seem remains of every iteration of the SDK
rm -R ~/.config/SailfishAlpha*
# created if you deploy an app
rm -R ~/rpmbuild
```

If you don't need VirtualBox anymore, mount the diskimage[47] and start the `VirtualBox_Uninstall.tool`, a bash script that removes all components form your system.

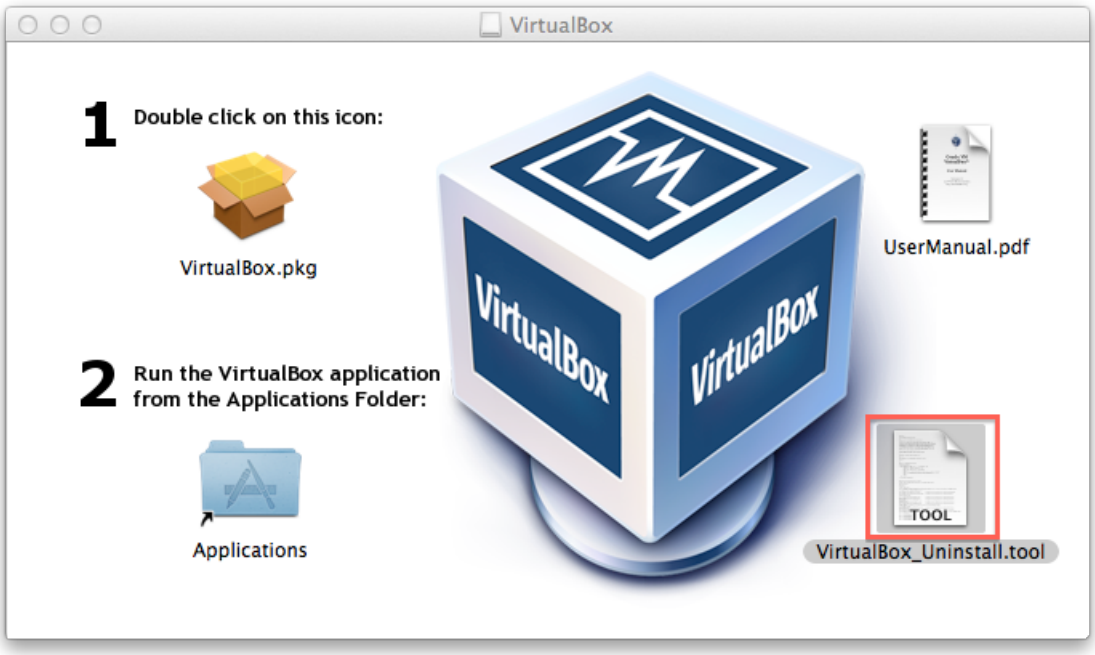


Figure 116: Uninstall of VirtualBox.

15.0.5 Windows

- Go to the control panel, choose "programs and features" and uninstall the SailfishOS SDK.
- Uninstall the VirtualBox from the same window if you don't need it anymore.

[47] You still have that,don't you?

15.0.6 Linux

Sorry, as for installation: until now I can not provide information here.

16 Thanks

Of course a big thanks goes out to everybody at Jolla. You are `#unlike <3`

Thanks go to David Greaves for convincing me to use another markup language[48].

A big thank you goes to Juergen Bocklage-Ryannel and Johan Thelin for creating such a wonderful online resource about QML[qml01].

Artem Marchenko takes care of the missing hello world, thank you!

17 Appendix

17.1 mb2 - bash script

```
#!/bin/bash
# Copyright (C) 2013 Jolla Ltd.
# Contact: David Greaves <david.greaves@jollamobile.com>
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:
#
# - Redistributions of source code must retain the above copyright
#   notice, this list of conditions and the following disclaimer.
# - Redistributions in binary form must reproduce the above copyright
#   notice, this list of conditions and the following disclaimer in
#   the documentation and/or other materials provided with the
#   distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
    "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
   TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
   PURPOSE
```

[48] Yes, it's still not converted but hey, I am already persuaded

```
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR
    CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
    OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
    BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
    WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
    OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
     OF THE
# POSSIBILITY OF SUCH DAMAGE.

short_usage() {
    cat <<EOF
usage: $ME [-t <target>] [-s <specfile>] [-d <device>] [-p <
    projectdir>]
           build [-d] [-j <n>] [<args>] | qmake [<args>] | make [<
    args>] | ssh <args>
           install [<args>] | rpm [<args>] | deploy <args> | run <
    args>
       $ME --help
EOF
    # exit if any argument is given
    [ -n "$1" ] && exit 1
}

usage() {
    short_usage
    cat <<EOF

  Executes a subset of build commands in the context of an rpmbuild.
  Typically called from QtCreator to perform qmake/make phases of a
    project.
  Note that any other build steps in the .spec file will also be run.

  <specfile> will be looked for in the current rpm/ dir. If there is
  more than one it must be provided.

  CWD is used as a base dir for installroot/ and RPMS/ to allow for
  shadowbuilds

  $ME is aware of spectacle and will update the spec file if there is
  an obvious yaml file which is newer.

  $ME build [-d] [-j <n>] [<args>]
                      : runs rpmbuild for the given spec file in the
```

```
                        given sb2 target. Produces an rpm package.
                      : -d     enable debug build
                      : -j <n> use only 'n' CPUs to build
                      : can use -s -t -p

  $ME qmake [<args>] : runs qmake in the 'build' phase
                        Note that this also verifies target
                        build dependencies are up to date
                      : can use -s -t -p

  $ME make [<args>]  : run make in the 'build' phase
                      : can use -s -t -p

  $ME deploy --zypper|--pkcon|--rsync
                      : runs the install or rpm-creation phase and
   then
                        copies/installs the relevant files to the
   device
                      : can use -s -t -p -d

  $ME run|ssh [<args>] : runs a command (on device if --device given)
   ;
                          intended for running gdb and a gdb server
                      : can use -s -t -p -d

  $ME install [<args>] : runs the 'install' phase to install to
   $buildroot
                      : can use -s -t -p
  $ME rpm [<args>]   : runs the install & rpm-creation phases
                      : can use -s -t -p


  -t | --target      : specify the sb2 target to use
  -d | --device      : specify the device
  -p | --projectdir  : when running shadow build/deploy from another
   dir
  -s | --specfile    : if the specfile is not in rpm/*.spec and
   cannot be found using -p

EOF
}

fatal() {
    echo "Fatal: $*"
    exit 1
}

assert_spec_supports_mb2() {
    if ! grep "define qtc_qmake" "$spec" >/dev/null 2>&1 ; then
```

```
  fatal "This specfile does not have the qtc_* macros defined"
    fi
}

try_to_make_spec() { # Tries to create a missing spec
    [[ -f "$1" ]] && return # It's not missing
    yaml=$(dirname "$1")/$(basename "$1" .spec).yaml
    [[ -f "$yaml" ]] || return # No yaml
    ANSI_COLORS_DISABLED=1 specify -n -N "$yaml"
}

try_to_make_spec_from_yaml() {
    # Tries to create a missing spec from a given yaml
    try_to_make_spec $(dirname "$1")/$(basename "$1" .yaml).spec
}

ensure_spec_newer_than_yaml() {
    yaml=$(dirname "$spec")/$(basename "$spec" .spec).yaml
    [[ -f "$yaml" ]] || return # User has decided not to use yaml
    if [[ "$yaml" -nt "$spec" ]]; then # -nt is newer than
  ANSI_COLORS_DISABLED=1 specify -n -N "$yaml"
    fi
}

verify_target_dependencies() {
    rpmspec --query --buildrequires "$spec" | \
        xargs --no-run-if-empty sb2 -t "$target" -m sdk-install -R
   zypper --non-interactive in
}

get_spec_tag() {
    rpmspec --query --srpm --queryformat="$1" "$spec"
}

# Helper to read XML
read_dom () {
    local IFS=\>
    read -d \< ENTITY CONTENT
    local RET=$?
    TAG_NAME=${ENTITY%% *}
    ATTRIBUTES=${ENTITY#* }
    return $RET
}

# This slurps the XML and converts tags like <subnet> to
   $device_subnet
# Also sets device_name and device_type from the attributes
get_device() {
    local FOUND_DEVICE=
```

```
    local IN_DEVICE=
    local maintag=
    while read_dom; do
  case $TAG_NAME in
      device )
    maintag=$TAG_NAME
    eval local $ATTRIBUTES
    if [[ "$name" == "$1" ]] ; then
        FOUND_DEVICE=1
        IN_DEVICE=1
        device_name="$name"
        device_type="$type"
    else
        IN_DEVICE=
    fi
    ;;
      engine )
    maintag=$TAG_NAME
    eval local $ATTRIBUTES
    ;;
      mac|index|subnet|ip|sshkeypath )
    # Don't process and store nested tags if we're in
    # device with wrong name
    if [[ "$maintag" == "device" ]] && [[ $IN_DEVICE != 1 ]]; then
        continue
    fi
    eval ${maintag}_$TAG_NAME="$CONTENT"
    ;;
  esac
    done
    # Set this up as it's useful
    if [[ "$device_subnet" ]]; then
  device_ip="$device_subnet".$device_index
    fi
    if [[ "$FOUND_DEVICE" == 1 ]]; then return 0; else return 1; fi
}

rsync_as() {
    local user=$1;shift
    local key="$(dirname "$DEVICES_XML")/${device_sshkeypath}/$user"
    [[ -f "$key" ]] || fatal "No key for user $user on $device_name
   given in devices.xml"
    RSYNC_RSH="ssh -F /etc/ssh/ssh_config.sdk -l $user -i $key" rsync
     "$@"
}

ssh_as() {
    local user=$1;shift
    local key="$(dirname "$DEVICES_XML")/${device_sshkeypath}/$user"
```

```
    [[ -f "$key" ]] || fatal "No key for user $user on $device_name
   given in devices.xml"
    ssh -F /etc/ssh/ssh_config.sdk -i $key -l $user $device_ip $@
}

cd_to_spec_setup_dir() {
    _basedir=$(pwd)
    local setup_dir=$(grep -Po '%setup.*-n.*' "$spec" | cut -s -f2 -d
   '/')
    if [ -n "$setup_dir" ]; then
  # in case we got a spec %{tag} out of this, try to expand it
  setup_dir=$(get_spec_tag "$setup_dir")
  cd "$setup_dir"
    fi
}

run_build() {
    # intended to provide mb build behaviour
    verify_target_dependencies

    sed -e '/^%patch/d' "$spec" > "$spec".$$

    (
  cd_to_spec_setup_dir;
  rm -f $_basedir/RPMS/*;
  eval sb2 -t $target rpmbuild --build-in-place \
      $BUILD_DEBUG \
      --define \"_smp_mflags -j$BUILD_JOBS\" \
      --define \"_rpmdir $_basedir/RPMS \" \
      --define \"_sourcedir $_basedir/rpm \" \
      --define \"_rpmfilename %%{name}-%%{version}-%%{release}.%%{
   arch}.rpm \" \
      --buildroot=\"$buildroot\" \
      --dobuild \
      --doinstall \
      --dobinary \
      --docheck \
      \"$spec\".$$
    )
    rm -f "$spec".$$
}

run_qmake() {
    if [[ "$spec" ]]; then
  # This is a good time to verify the target dependencies as per mb
  verify_target_dependencies
  (
      cd_to_spec_setup_dir;
      eval sb2 -t $target rpmbuild --build-in-place \
```

```
    --dobuild \
    --define \"noecho 1 \" \
    --define \"qtc_builddir $_basedir \" \
    --define \"qtc_make true ignoring make\" \
    --define \"qtc_qmake5  %qmake5 $@\" \
    --define \"qtc_qmake  %qmake $@\" \
    \"$spec\"
  )
    else
  sb2 -t $target qmake "$@"
    fi
}

run_make() {
    if [[ "$spec" ]]; then
  (
      cd_to_spec_setup_dir;
      eval sb2 -t $target rpmbuild --build-in-place \
    --dobuild \
    --define \"noecho 1 \" \
    --define \"qtc_builddir $_basedir \" \
    --define \"qtc_qmake5 true ignoring qmake\" \
    --define \"qtc_qmake true ignoring qmake\" \
    --define \"qtc_make  make  %{?_smp_mflags} $@\" \
    \"$spec\"
  )
    else
  sb2 -t $target make "$@"
    fi
}

run_install() {
    # Install to buildroot which should be rsync'ed to /opt/sdk/
   $package on device
    (
  cd_to_spec_setup_dir;
  eval sb2 -t $target rpmbuild --build-in-place \
      --define \"noecho 1 \" \
      --define \"qtc_builddir $_basedir \" \
      --define \"_sourcedir $_basedir/rpm \" \
      --buildroot=\"$buildroot\" \
      --doinstall \
      \"$spec\"
    )
}

run_rpm() {
    (
  cd_to_spec_setup_dir;
```

```
  rm -f $_basedir/RPMS/*;
  eval sb2 -t $target rpmbuild --build-in-place \
      --define \"qtc_builddir $_basedir \" \
      --define \"_rpmdir $_basedir/RPMS \" \
      --define \"_sourcedir $_basedir/rpm \" \
      --define \"_rpmfilename %%{name}-%%{version}-%%{release}.%%{
    arch}.rpm \" \
      --buildroot=\"$buildroot\" \
      --doinstall \
      --dobinary \
      --docheck \
      \"$spec\"
    )
}

run_deploy() {
    [[ "$device_type" ]] || fatal "deploy must have a valid --device"
    local fail_text="deploy must use one of --pkcon, --rsync or --
    zypper"
    [[ -z ${1:-} ]] && fatal $fail_text

    while [[ $1 ]]; do
  case "$1" in
      "--pkcon" ) shift
    run_rpm
    rpms=$(find RPMS -name \*rpm | grep -v -- "-debug")
    rsync_as root -av ${rpms} $device_ip:/root/RPMS/
    ssh_as root pkcon --plain --noninteractive install-local ${rpms}
    ;;
            "--zypper" ) shift
    run_rpm
    rpms=$(find RPMS -name \*rpm | grep -v -- "-debug")
    rsync_as root -av ${rpms} $device_ip:/root/RPMS/
    ssh_as root zypper --non-interactive in -f ${rpms}
    ;;
            "--rsync" ) shift
    user=$deviceuser
    run_install
    name=$(get_spec_tag "%{name}")
    rsync_as $user -av ${buildroot}/. $device_ip:/opt/sdk/$name
    ;;
      *)
    fatal $fail_text ;;
  esac
    done
}

ME=$(basename $0)
target=""
```

```
pkgdir="./rpm"
DEVICES_XML=/etc/mersdk/share/devices.xml
deviceuser=nemo

# Virtualbox environment will install in this hardcoded location
if [[ -f /etc/mer-sdk-vbox ]]; then
    buildroot=/home/deploy/installroot
else
    buildroot=$(pwd)/installroot
fi

while [[ "$1" ]]; do
    case "$1" in
        "-t" | "--target") shift
      target="$1"; shift
      ;;
        "-d" | "--device") shift
      device="$1"; shift
      get_device "$device" < $DEVICES_XML || fatal "'$device' not
   found in devices.xml"
      ;;
        "-p" | "--projectdir") shift
      projdir="$1"; shift
      pkgdir="$projdir"/rpm
      [[ -d "$projdir" ]] || fatal "'$projdir' is not a directory"
      ;;
        "-s" | "--specfile" ) shift
      spec="$1"; shift
      try_to_make_spec "$spec"
      [[ -f "$spec" ]] || fatal "'$spec' doesn't exist (and couldn't
   be made from a yaml)"
      ;;
  install | rpm | deploy | build )
      needspec=1;
      break 2 ;;
  qmake | make | run | ssh )
      break 2 ;;
  *)
      usage
      exit 1
            ;;
    esac
done

if [[ ! "$target" ]]; then
    if [[ -f ~/.scratchbox2/config ]]; then
  . ~/.scratchbox2/config
  target=$DEFAULT_TARGET
    fi
```

```
    [[ "$target" ]] || fatal "You must specify an sb2 target or have
    a default configured"
fi

[[ -d ~/.scratchbox2/$target ]] || fatal "$target is an invalid sb2
    target"

# spec rules are complex:
#  a .spec is required for some but not all operations
#  if -s is given then
#    if it does not exist then specify tries to make it
#    if it exists it will be used
#  if there is a rpm/*.spec then that is used
#  if there is a rpm/*.yaml then a spec is made and used

if [[ ! "$spec" ]]; then
    numspec=$(ls "$pkgdir"/*spec 2>/dev/null | wc -l)
    if [[ $numspec -gt 1 ]]; then
  [[ $needspec ]] && fatal "Too many spec files - please use -s to
    identify which to use"
  echo "Too many spec files - not using any. Use -s to identify a
    specific one"
  # spec is not set
    fi

    if [[ $numspec -eq 0 ]]; then
  # No spec, try to find a yaml
  numyaml=$(ls "$pkgdir"/*yaml 2>/dev/null | wc -l)
  if [[ $numyaml -eq 1 ]]; then
      try_to_make_spec_from_yaml $(ls "$pkgdir"/*yaml)
      numspec=$(ls "$pkgdir"/*spec 2>/dev/null | wc -l)
  else
      [[ $needspec ]] && fatal "No spec file found in '$pkgdir/' and
    couldn't make one from a yaml"
  fi
    fi

    if [[ $numspec -eq 1 ]]; then
  spec=$(ls "$pkgdir"/*spec)
    else
  # this is because we did try_to_make_spec_from_yaml and failed
  [[ $needspec ]] && fatal "No spec file found in '$pkgdir/' and
    couldn't make one from a yaml"

    fi
fi

# Now if there is a spec given, make sure it is up-to-date
if [[ "$spec" ]]; then
```

```
    # turn 'spec' into an absolute path
    spec=$(readlink -f "$spec")
    ensure_spec_newer_than_yaml
fi

case "$1" in
    qmake | make | install | rpm | deploy )
  cmd=run_$1
  if [[ "$spec" ]]; then
      assert_spec_supports_mb2
  fi
        shift
        ;;
    build )
  cmd=run_$1; shift
  BUILD_DEBUG='--define "debug_package %{nil}" '
  BUILD_JOBS=$(getconf _NPROCESSORS_ONLN)
  while [[ "$1" ]]; do
      case "$1" in
    -d|--enable-debug) shift
        BUILD_DEBUG= ;;
    -j*)
        # support giving -j with and without space between
        # it and the 'n'
        if [ ${#1} -gt 2 ]; then
      BUILD_JOBS=${1:2}; shift
        else
      [ -z "$2" ] && short_usage quit
      BUILD_JOBS="$2"; shift 2;
        fi
        ;;
    *)
        break
        ;;
      esac
  done
        ;;
    run | ssh ) shift
  if [[ "$device" ]]; then
      cmd="ssh_as $deviceuser"
  else
      cmd="eval"
  fi
        ;;
    *)
  short_usage quit
  ;;
esac
```

```
$cmd "$@"
```

../media/sdk/mb2

[language=bash]

References

[jolla01] Jolla ltd., based in Finland - the inventors of the Jolla smartphone
http://jolla.com

[jolla02] Jolla Harbour - the place for your apps
https://harbour.jolla.com

[sailfishos01] SailfisoS - the operating system driving the Jolla smartphone
https://sailfishos.org

[sailfishos2] Quick introduction into development with the SailfishOS SDK
https://sailfishos.org/develop.html

[sailfishos3] SailfishOS SDK overview
https://sailfishos.org/develop-overview-article.html

[sailfishos4] SailfishOS open source code
http://releases.sailfishos.org/sdk/

[sailfishos5] Unofficial Sailfish OS third party open source apps collection
https://github.com/sailfishapps

[sailfishos6] SailfishOS, Packaging Applications for Distribution
https://sailfishos.org/develop-packaging-apps.html

[vbox01] VirtualBox, a virtualization platform from Oracle
https://www.virtualbox.org/wiki/Downloads

[vbox02] Guest shows IDE/SATA errors for file-based images on slow host file system
http://www.virtualbox.org/manual/ch12.html#idp60327840

[hc01] hardcodes, that's my nickname and my website
http://www.hardcodes.de

[hc02] This document on Github
https://github.com/hardcodes/developwithsailfishos.git

[hc03] Alternative Icon for the QtCreator inside of the SailfishOS SDK
http://blog.hardcodes.de/articles/68/sailfish-os-icon

[hc04] Just the PDF version of this document
http://hardcodes.de/SailfishOS/Developing-with-SailfishOS.pdf

[qt01] The Qt Project.
https://qt-project.org

[qt02] QtCreator.
https://qt-project.org/doc/qtcreator-2.8/

[qt03] qmake manual.
http://qt-project.org/doc/qt-4.8/qmake-manual.html

[qt04] Meta-Object Compiler (MOC)
http://qt-project.org/doc/qt-4.8/moc.html

[qt05] QML
http://qt-project.org/doc/qt-5.0/qtqml/qtqml-index.html

[qt06] QtQuick
http://qt-project.org/doc/qt-5.0/qtquick/qtquick-index.html

[qt07] QtCreator, Adding New Custom Wizards
http://qt-project.org/doc/qtcreator-2.8/creator-project-wizards.html

[wiki01] Wikipedia, Model–view–controller (MVC)
http://en.wikipedia.org/wiki/Model\T1\textendashview\T1\textendashcontroller

[wiki02] Wikipedia, Scratchbox2
http://en.wikipedia.org/wiki/Scratchbox2

[wiki03] RPM - Redhat Package Manager
http://en.wikipedia.org/wiki/RPM_Package_Manager

[sb2] Scratchbox2 homepage
https://maemo.gitorious.org/scratchbox2

[mer01] Mer Wiki, Platform SDK and SB2
https://wiki.merproject.org/wiki/Platform_SDK_and_SB2

[mer02] Mer Wiki, SailfishOS SDK hacks
https://wiki.merproject.org/wiki/Sailfish_SDK_Hacks

[mer03] Mer Project, start page
http://merproject.org

[mer04] Mer SDK on VirtualBox/Design https://wiki.merproject.org/wiki/SDK_on_VirtualBox/Design

[mer05] Mer Wiki, spectacle
https://wiki.merproject.org/wiki/Spectacle

[yaml01] yaml specs
http://www.yaml.org/spec/

[ex01] Jolla details on eLinux
http://elinux.org/Jolla

[sdc01] SmartDevCon
http://smartdevcon.eu

[gh01] The missing HelloWorld. Wizard included by Artem Marchenko
https://github.com/amarchen/helloworld-pro-sailfish

[suse01] openSUSE Portal:Zypper
http://en.opensuse.org/Portal:Zypper

[nm01] Nmap, network security scanner
http://nmap.org

[irc01] A Short IRC Primer
http://irchelp.org/irchelp/ircprimer.html

[qml01] QML Book - free online resource about QML
http://qmlbook.org/

[rpm01] Fedora project - how to build packages
https://fedoraproject.org/wiki/How_to_create_an_RPM_package

[rpm02] Fedora project - PackageKit CLI (pkcon)
http://docs.fedoraproject.org/en-US/Fedora/14/html/Software_Management_Guide/PackageKit_CLI.html

[fd01] FreeDesktop - pkg-config guide
http://people.freedesktop.org/~dbn/pkg-config-guide.html