Embedded Edition

February 2017 – Based on Qt 5.5

Contents

Qt Embedded	Embedded Configurations, Feature Management, Memory Footprint, Embedded Tool Chains, Yocto Project, Cross-Compilation, Deployment
Qt GUI Integration	QPA Plugin, Screen, Window, Backing Store, and GL Context, GUI Event System Integration, Integration Classes, Themes
Qt Enterprise Virtual Keyboard	Usage, Customization
Qt Serial Bus	Serial Bus Usage, Backends
Boot2Qt	Boot2Qt, Embedded App Creation, Building, Debugging, and Deployment, Build System Customization

Objectives

- To learn essential Qt libraries for application engine development
  - Qt programming on embedded targets
  - Qt Enterprise Embedded AKA Boot2Qt

- Any questions at any point – please do not hesitate to ask!

Qt Embedded

Contents

- Embedded Configurations
- Feature Management
- Memory Footprint
- Embedded Tool Chains
- Yocto Project
- Cross-Compilation
- Deployment

Qt Embedded

- Prior Qt5, Qt Embedded was based on Qt Window System (QWS)
- Since Qt5, no Qt-specific window system
  - New compositor Technology Preview in Qt 5.6

- Qt Embedded means building Qt to embedded targets
- Some platforms require more adaptation than others
  - No Posix APIs
  - No processes

Building Qt Libraries for Embedded Platforms

1. Create a target toolchain
   - Target image and rootfs created as well
2. Configure Qt
   - Write/edit platform-specific **`MKSPECS`**-file
   - Configure required features, add/remove features
3. Build Qt libraries using the target tool chain
4. Deploy Qt libraries to your target device

- Boot2Qt helps in all phases
  - Provides pre-built target image, Qt libraries, and tool chain
  - Just deploy the target image and start developing Qt programs

1. Create a Target Toolchain

- Package to cross-compile Linux + Qt libraries + other SW to the embedded target
    - Cross-compiler, linker, possibly a debugger
    - Boot code (u-boot)
    - Root file system (rootfs) including
        - Linux kernel, Board Support Package (BSP) for the HW in question

- Time consuming process
    - Build the toolchain tool with all the dependencies
    - Build the bootloader, kernel, and root file system with optimal configurations

    - Optimal configuration for the performance?
    - Optimal configuration for minimal memory footprint?
    - Silicon vendors may provide useful configurations for the HW platform

Embedded Toolchain

- A tool to cross-compile SW in some host to the target board
  - Compiler, linker, assembler tool, C library
  - CodeSourcery – Eclipse-based IDE and GNU toolchain for numerous target architectures
  - Linaro – optimized toolchains for recent ARM CPUs (Cortex A8, A9) implemented partially by CodeSourcery employees
  - DENX Embedded Linux Development Kit (ELDK) – Cross compilation tools + U-Boot, Linux kernel + drivers for ARM, PowerPC, and MIPS processors
  - ScratchBox – toolchanis for x86 and ARM target architectures
- Toolchain Building System
  - Build a toolchain from the sources and possibly build the whole target operating system ready to be flashed
  - Buildroot – complete build system, based on Linux kernel configuration system
  - Crosstool-NG – similar to Buildroot, targeting at easier configuration
    - Both toolchains support a wide variety of target architectures
  - Bitbake - tool to build the complete distribution (Ångström), used by OpenEmbedded

Linaro

- Open organization focused on improving Linux on ARM
  - http://www.linaro.org/about/

- Linaro Linux Kernel
  - Modern, optimized
  - Supports thumb instructions

- Linaro Toolchain
  - GCC-based compiler, various versions of GCC supported
  - GDB
  - QEMU to run (emulate) ARM binaries

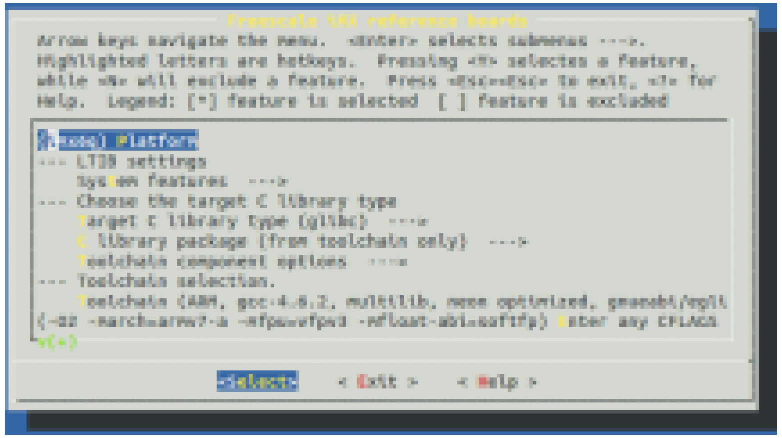
Crosstool-NG

- Crosstool New Generation (NG)
  - http://crosstool-ng.org/
- Tool to build toolchains from the scratch
  - Latest features and optimizations
  - The toolchain may be used to build the root file system (**buildroot** tool)
- Support for, e.g. Linaro toolchain
- Well-known kernel like **menuconfig** interface to select the configuration (target settings, compiler configuration, glib, Qt libraries etc.)

- Though easy to configure, embedded developer must know what to do
  - What are the latest SW packages (may be needed to setup and configure manually)
  - What is the right toolchain configuration to use? All toolchains do not support hard floating points, for example

LTIB – Linux Target Image Builder

- Open source project for creating Linux BSPs and images maintained at Savannah
  - http://savannah.nongnu.org/projects/ltib
- BSP (Board Support Packages)
  - HW-specific boot and driver code
- Concept similar to Buildroot
  - To build the root file system
- Over 200 packages and BSPs for Freescale CPUs
- For iMX6 (ARM), uses Linaro toolchain
- First, may be time consuming to setup but then rather easy to use
  - Hard coded paths in Perl scripts (need to be fixed)
  - Out of the date packages – need to update by manually editing repo references, e.g. Qt 5.x libraries



```
                 Freescale iMX reference boards
 Arrow keys navigate the menu.  <Enter> selects submenus --->.
 Highlighted letters are hotkeys.  Pressing <Y> selects a feature,
 while <N> will exclude a feature.  Press <Esc><Esc> to exit, <?> for
 Help.  Legend: [*] feature is selected  [ ] feature is excluded

 (imx6q) Platform
 --- LTIB settings
     System features  --->
 --- Choose the target C library type
     Target C library type (glibc)  --->
     C library package (from toolchain only)  --->
     Toolchain component options  --->
 --- Toolchain selection.
     Toolchain (ARM, gcc-4.6.2, multilib, neon optimized, gnueabi/eglibc
 (-O2 -march=armv7-a -mfpu=vfpv3 -mfloat-abi=softfp) Enter any CFLAGS
 v(+)

              <Select>    < Exit >    < Help >
```

Open Embedded Project

- Open source project, providing tools to build a complete Linux distribution for embedded systems
  - http://www.openembedded.org
- Over 1,000 packages
- Based on the layers on the top of OE-Core
  - Base layer for recipes (details of pieces of SW), classes (build info) and other files (configurations)
  - More than 7,500 recipes exist covering 300 machines and 200 distros
  - Support for ARM, x86, x86-64, PowerPC, and MIPS
  - Distro-less, though standalone image may be built
  - Split out from Poky distro (Yocto project)
- Another layer meta-openembedded
  - Contains items shared by multiple layers, but do not fit into the OE-Core

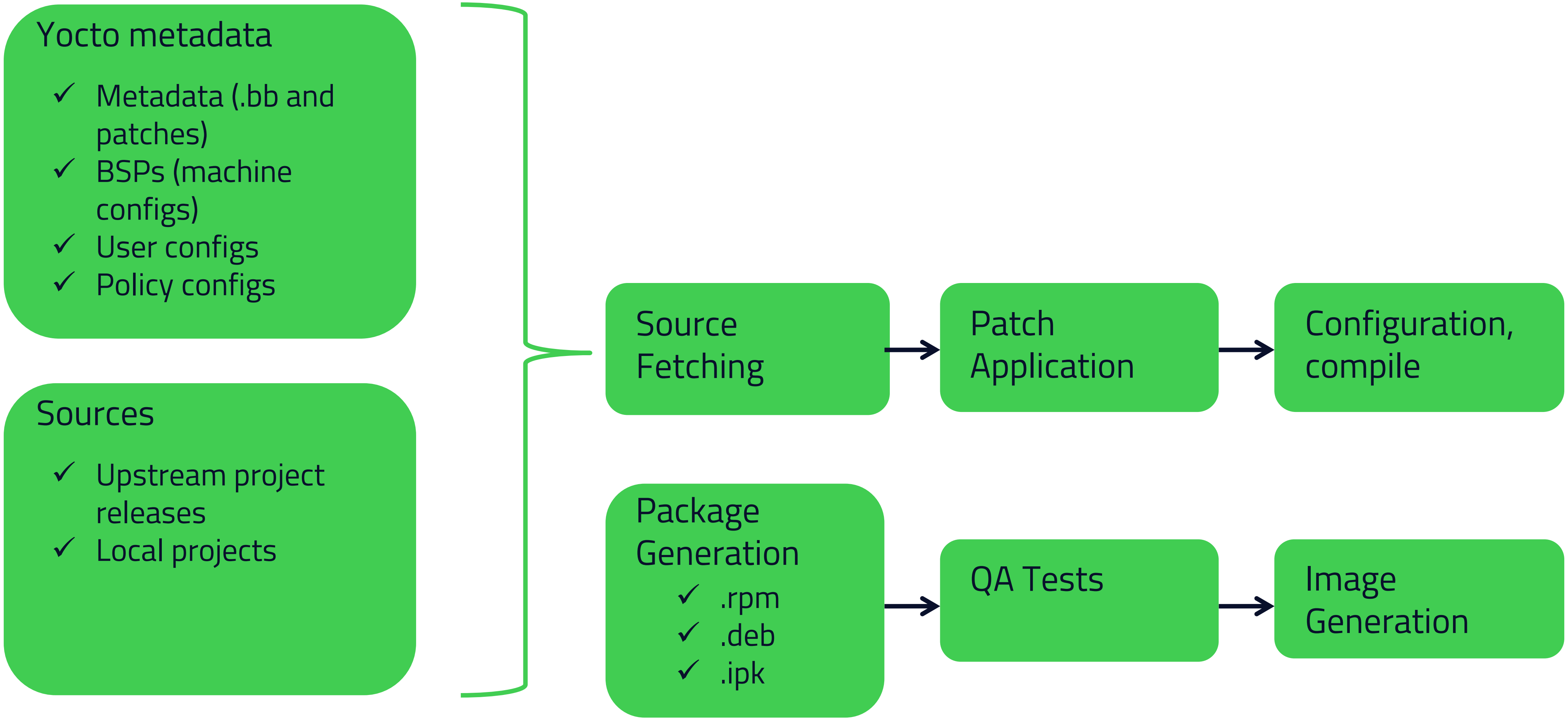
Ångström

- Distro on top of oe-core and meta-openembedded
  - http://www.angstrom-distribution.org
- Very small minimum memory footprint 4MB
  - Compare to Qt core and GUI libs, which alone have approximately the same size
- The buildsystem uses various components from Yocto
  - Bitbake cross-compiler
  - Application and BSP layers
- Widely used on TI-based embedded boards, like BeagleBoard and PandaBoard

Yocto Project

- Open source project and Linux Foundation workgroup, providing templates, tools, packages and so on to create embedded Linux distros
  - https://www.yoctoproject.org
- Open source build system and toolchain called Poky
  - OE-Core is one Poky branch
- Automates the fetch of source packages
  - No script updating
  - Package details defined in recipes
  - Recipes easy to add and replace
- Package format and architecture agnostic
- Core build tool (Bitbake) and metadata syntax shared with Open Embedded project
- Used by Qt Enterprise Embedded

Yocto Development Environment



Yocto metadata

- ✓ Metadata (.bb and patches)
- ✓ BSPs (machine configs)
- ✓ User configs
- ✓ Policy configs

Sources

- ✓ Upstream project releases
- ✓ Local projects

Source Fetching → Patch Application → Configuration, compile

Package Generation

- ✓ .rpm
- ✓ .deb
- ✓ .ipk

→ QA Tests → Image Generation

Yocto Toolchain Practical Steps High-Level Description

- Step 1: Get Yocto

```
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git checkout -b dizzy origin/dizzy
```

- Step 2: Initialize the build environment

```
$ source oe-init-build-env buildDir
```

  - Creates, e.g. **`conf/bblayers.conf`** and **`conf/local.conf`**
  - The first file defines the layers used
  - The latter one defines user configuration

- Step 3: Configure the **`local.conf`** file
  - Located in the **build** folder
  - Look at the reference documentation for syntax and variables
  - For example, define your target platform `MACHINE ?= "beaglebone"`

- Step 4: Create the image

```
$ bitbake core-image-minimal
```

  - May take several hours

Layers (OE-Core)

- Layers define the SW packages you want to include into your image
- Define the layers you wish to use in your build folder **`conf/bblayers.conf`** file
- Download or create new layers
  - E.g. for toradex from **git://git.toradex.com/meta-toradex.git**
  - Look at examples at Qt Enterprise Embedded folder **`<installation folder>/Boot2Q/sources/b2qt-yocto-meta`**
- Layers contain
  - Recipes
  - Configuration
  - Classes

Some example layers

- openembedded-core
- BSPs
  - meta-fsl-arm
  - meta-raspberrypi
- Distros
  - meta-angstrom
  - meta-yocto (Poky)
- SW
  - meta-gstreamer10
  - meta-go, meta-java
- Misc
  - meta-linaro

Configuration

- Configuration files (**`*.conf`**)
- Define variables used by build scripts (recipes)
- Compare to **qmake** variables, some similarities in the syntax (shell-like)
- `DISTRO ?= “poky”`
- Look at the syntax details in Yocto reference manual and usage details in the sample files
- Configuration files
  - User configuration – **`local.conf`**
  - Build configuration – **`bitbake.conf`**
  - Machine configuration – **`emulator.conf`**
  - Distro configuration – **`b2qt.conf`**

Bitbake Classes

- Classes (**`*.bbclass`**)
- Provide information, which is useful to share between recipes
- Define the configuration scripts
- Refer to variables in configuration files
- For example, how to configure generic recipe agnostic tests

Recipes

- Provides details about particular pieces of SW (**`*.bb`**)
  - Local and remote repositories
  - Patches
  - Configuration
  - Build scripts
  - Package options (**`.deb`**, **`.rpm`**, **`.idk`**)

- Provides additional SW-specific variables and script functions

- Used by Bitbake tool

Building

- Targets are defined in recipes
- Building itself straightforward

```
$ bitbake core-image-minimal
```

  - File **`core-image-minmal.bb`** found in **`poky/meta/recipes-core/images`** folder
  - The `core-image-minimal` version could be defined in distribution configuration options

- Target may have additional features defined in the recipe file

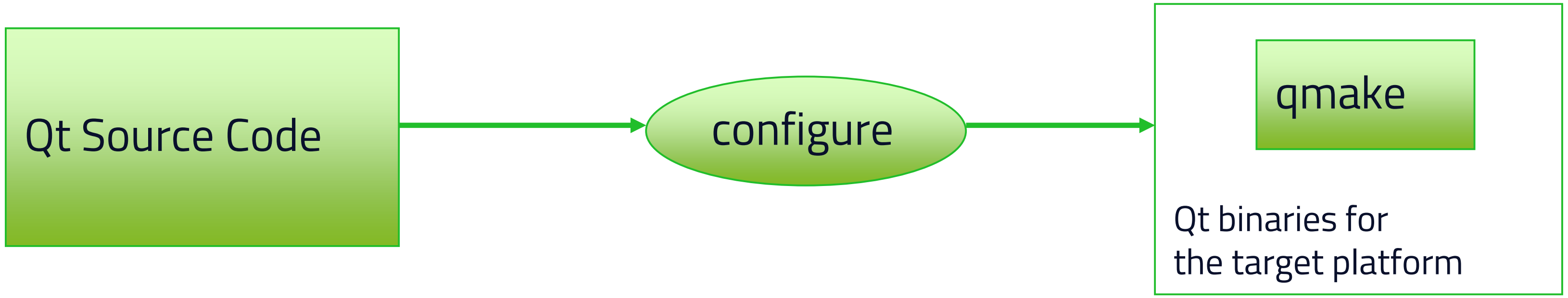
- A core image may for example include
  - Boot code
  - Qt libraries

2. Configure Qt

- **`<QtSrc>/qtbase/configure`** is a tool for configuring/building Qt itself for the target platform
  - Creates Makefiles for the modules
  - Builds e.g. platform-specific tools: **`qmake`**, **`uic`**, **`rcc`**, **`moc`**

- Can be used to manage Qt features / memory footprint of Qt libraries

- Embedded configuration
  - `configure -device <device name>`
  - `make module-qtbase`
  - `make install`



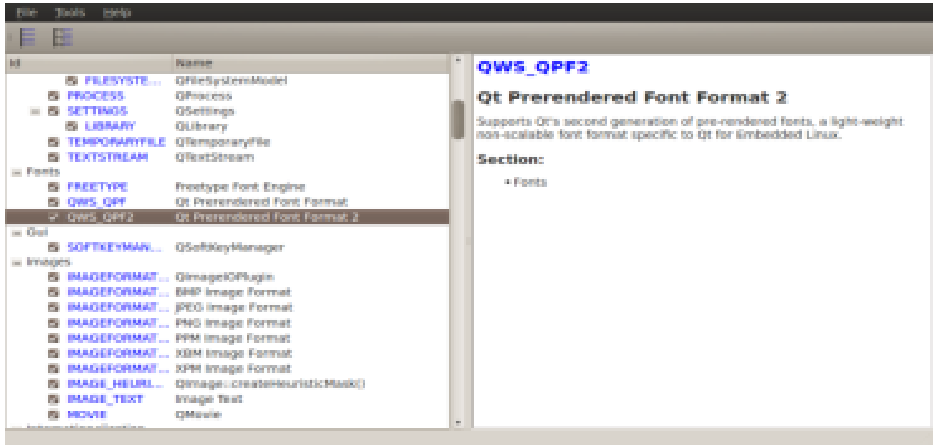
Useful Configuration Options

```
-device <device name>
-device-option CROSS_COMPILE=<CROSS COMPILER PATH>
-sysroot <sysroot path>
-qpa // Default QPA platform
-prefix <dir>
-pch / -no-pch // Pre-compiled headers
-feature / -no-feature // -no-feature-accessibility
-opengl <api> // es1, es2
-qconfig <custom> // Custom configuration in src/corelib/global/qfeatures.txt
-shared / static
-debug / release // -separate-debug-info –force-debug-info
-qt-<library name> / -system-<library name>
```

Toradex Configuration – Made by Build Scripts

```
./configure

-commercial
-confirm-license
-release
-device linux-imx6-g++
-device-option CROSS_COMPILE=/toolcahin/sysrootfs/armv7ahf-vfp-neon-poky-linux-gnueabi/usr/share
-sysroot /toolcahin/sysrootfs/armv7ahf-vfp-neon-poky-linux-gnueabi
-no-xcb
-nomake examples
-nomake tests
```

Managing Memory Footprint



Id	Name
FILESYSTE...	QFileSystemModel
PROCESS	QProcess
SETTINGS	QSettings
LIBRARY	QLibrary
TEMPORARYFILE	QTemporaryFile
TEXTSTREAM	QTextStream
Fonts	
FREETYPE	Freetype Font Engine
QWS_QPF	Qt Prerendered Font Format
QWS_QPF2	Qt Prerendered Font Format 2
Gui	
SOFTKEYMAN...	QSoftKeyManager
Images	
IMAGEFORMAT...	QImageIOPlugin
IMAGEFORMAT...	BMP Image Format
IMAGEFORMAT...	JPEG Image Format
IMAGEFORMAT...	PNG Image Format
IMAGEFORMAT...	PPM Image Format
IMAGEFORMAT...	XBM Image Format
IMAGEFORMAT...	XPM Image Format
IMAGE_HEURI...	QImage::createHeuristicMask()
IMAGE_TEXT	Image Text
MOVIE	QMovie

QWS_QPF2

Qt Prerendered Font Format 2

Supports Qt's second generation of pre-rendered fonts, a light-weight non-scalable font format specific to Qt for Embedded Linux.

Section:

- Fonts

- Create a custom configuration
- Simple way
  - `-no-feature-<feature> | -feature-<feature>`
  - `/src/corelib/global/qfeatures.txt`
  - `/src/corelib/global/qconfig-<minimal/small/medium/large>.h`
- Or use `qconfig` tool
  - Located in `/qttools/src/qconfig`

- Produces custom configuration file to configure Qt
  - Enable/disable features
  - Save under new name **`src/corelib/global/qconfig-myconfig.h`**
  - Run configure script with option `-qconfig myconfig`

Managing Memory Footprint

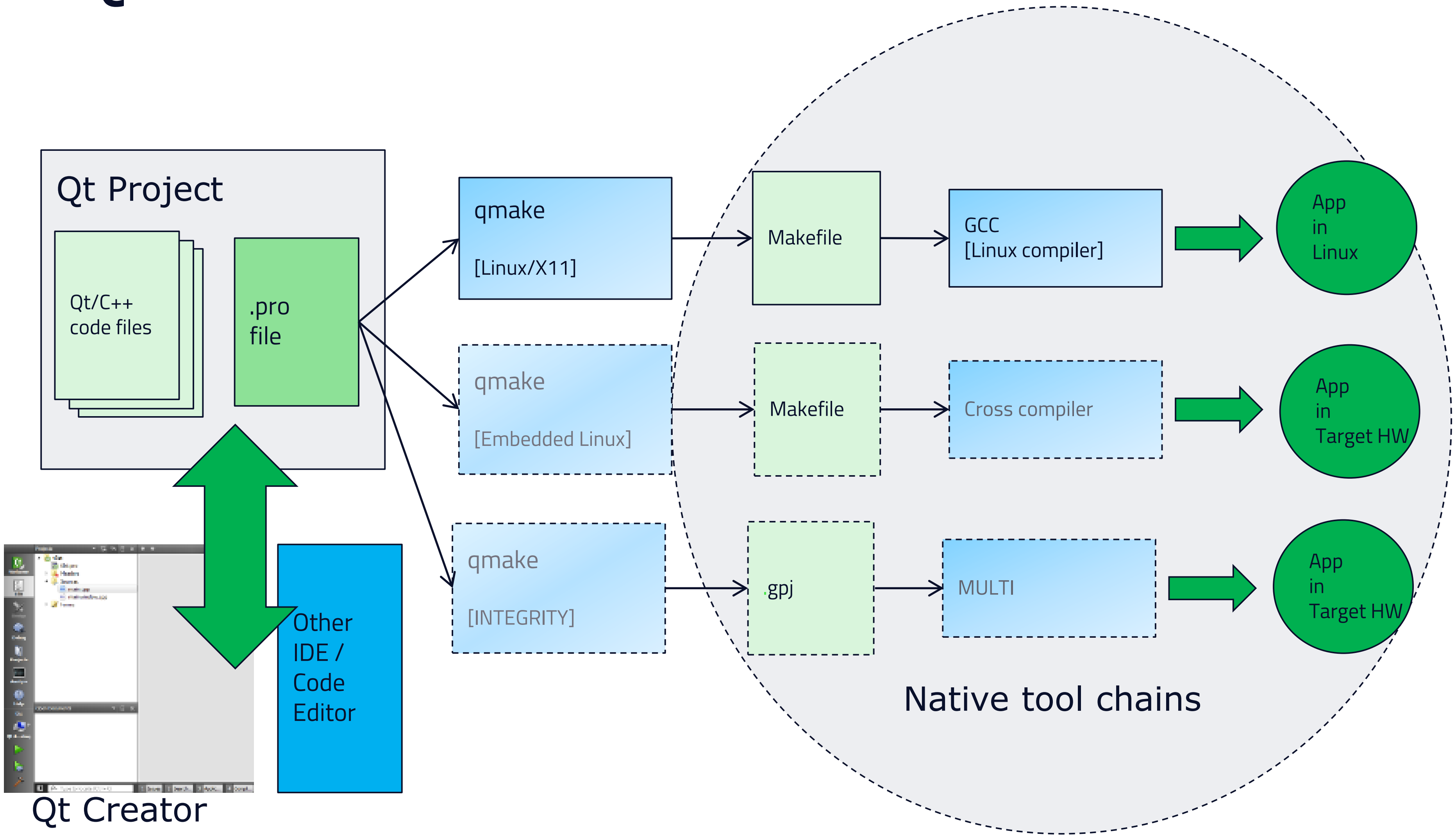
- Executable compression
  - Ultimate Packer for eXecutables- http://upx.sourceforge.net/
  - Obviously, there is a small performance trade-off for decompression
- Compilation options
  - Remove speed optimizations: `QMAKE_CXXFLAGS_RELEASE -= O2`
  - Enable size optimizations: `QMAKE_CXXFLAGS_RELEASE += Os`
- Do not forget to strip your final binaries

Cross-Compilation Configuration

- If needed, create a target-specific make specification in the folder **`<Qt src>/qtbase/mkspecs`**
  - Use a specification close to your platform (e.g., **`mkspecs/linux-arm-gnueabi-g++`**)
  - Sometimes provider by the chip vendor
- It is essential to define cross-compiler tools and build flags (**`qmake.conf`**)

```
QMAKE_CFLAGS       = -march=armv7-a -mfpu=neon -mfloat-abi=softfp
QMAKE_CXXFLAGS    = -march=armv7-a -mfpu=neon -mfloat-abi=softfp
QMAKE_CC          = arm-fsl-linux-gnueabi-gcc
QMAKE_CXX         = arm-fsl-linux-gnueabi-g++
QMAKE_LINK         = arm-fsl-linux-gnueabi-g++
QMAKE_LINK_SHLIB   = arm-fsl-linux-gnueabi-g++

# modifications to linux.conf
QMAKE_AR          = arm-fsl-linux-gnueabi-ar cqs
QMAKE_OBJCOPY     = arm-fsl-linux-gnueabi-objcopy
QMAKE_STRIP       = arm-fsl-linux-gnueabi-strip
load(qt_config)
```

3. Build Qt Libraries



4. Deployment – The Target Has Qt Libs

- The target platform has Qt libraries
  - Use `INSTALLS` variable in the **.pro** file to install any files
  - Use `QCoreApplication::addLibraryPath()/setLibraryPaths()` to add search path for plugins
  - `target.files = someFille *.qml qml.dir`
  - `installDestination = $$[QT_INSTALL_QML]/MyModule/SubName`
  - `target.path = $$installDestination`
  - `INSTALLS += target`

How to detect Qt version installed on the target?

- **`qmake -v`**
- Qt include folder
- Platform dependent tools
  - Linux: **`ldd`**
  - macos: `otool`
  - Windows: Dependency Walker (**depends** - http://www.dependencywalker.com)

Deployment – The Target Does not Have Qt Libs

- Create a static build
  - `configure -static -platform`
- Create a bundle manually
  - Copy the relevant Qt libs/plugins to your bundle
  - Write a script which sets relevant environment variables and launches your application

```
#!/bin/sh
 export LD_LIBRARY_PATH=`pwd`/qt_libs
 export QML2_IMPORT_PATH=`pwd`/qt_libs/qml
 export QT_QPA_PLATFORM_PLUGIN_PATH=`pwd`/qt_libs/plugins/platforms
 ./MyCoolApplication
```

Deployment – The Target Does not Have Qt Libs

- Use platform-dependent (OSX, Windows) deployment tools in **`QTDIR/bin`**
  - **`macdeployqt`**, **`windeployqt`**
  - Some options
    - `-no-plugins` (by default all release plugins will be added, if the corresponding Qt module used)
    - `-dmg` create a disk image in OSX
  - Third party libraries must still be manually copied to the bundle/added to the installation package
  - You may need to handle different architectures 32/64 bit, Intel/PowerPC etc.

- Create a custom binary installer
  - Used e.g. for the Qt SDK installers, and Qt Creator installer
  - Customizable
  - Offline or online

Creating Custom Installer

1. Create a *package directory structure*
2. Create a *configuration file*
3. Create a *package information file*
4. Create installer content and *copy* it to the package directory.
5. Use the **binarycreator** tool to create the *installer*

   The installer pages are created by using the information you provide in the configuration and package information file

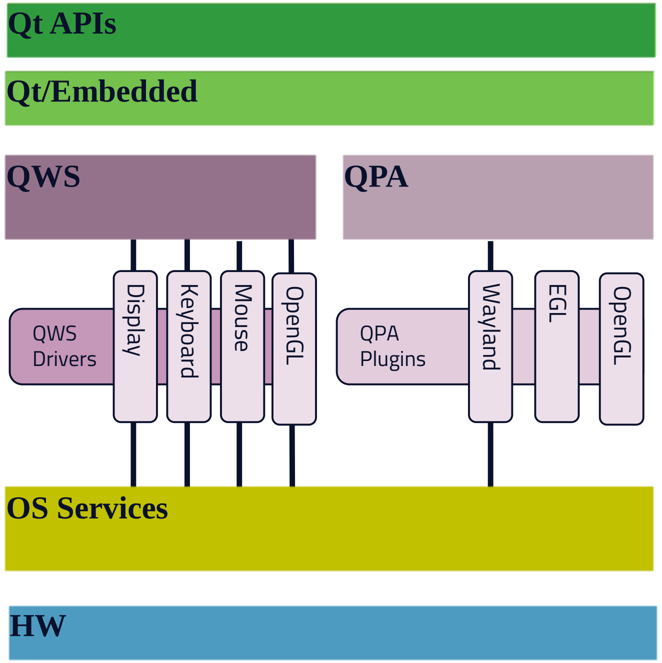
Qt GUI Integration
Platform Abstraction

Contents

- Platform Abstraction
- Screen, Window, Backing Store, GL Context
- GUI Events
- Essential QPA classes
- Themes

Qt (Embedded) Native Integration

- Prior Qt5, Qt Embedded was based on the lightweight window system called QWS
  - Replacing “heavy” X11 originally
- In Qt 5, QWS has been replaced by QPA – Qt Platform Abstraction, introduced in Qt 4.8
  - Not a window system, just a platform abstraction
- Other integrator related issues
  - CPU architecture
    - Atomic operations
  - Operating System
    - Requires libc, pthread, some math functions
    - QtCore runs well on a POSIX compliant OS/RTOS



QWS vs. QPA

- Qt Embedded is a lightweight window system
  - Applications write to the shared memory and QWS server composites the buffers
- Supports multiple processes and windows
- One process provides QWS server
  - Uses plug-ins (drivers) to manage input devices and screen output
  - Controls screen cursor appearance and screen saver
  - Hub for inter-process communication
- OpenGL-based acceleration is not always easy
  - PowerVR reference plugin exists

- QPA is not a window system
- A single platform dependent plug-in
  - DirectFB, LinuxFB, EGL, XCB, Windows, WinRT, iOS, Android, QNX, VxWorks
- Full OpenGL support
- Platform-specific window-system may be used
  - Or a plug-in may provide the window system (Wayland compositor)

Building and Using QPA Plugin

- Default QPA plugin may be define in configuration
  - `-qpa eglfs`
- The default platform may be replaced in run-time
  - Use the `–qpa` command line option or
  - Define the `QT_QPA_PLATFORM` environment variable
- Default QPA plugins
  - XCB for Linux
  - Windows for Windows
  - Cocoa for Mac
- Platform initialized by the `QGuiApplication`
  - Uses `QPlatformIntegrationFactory` to load the QPA plugin (`qLoadPlugin1()`)
  - The plugin uses `QPlatformIntegrationPlugin` to instantiate a `QPlatformIntegration` sub-class

EGLFS Plugin

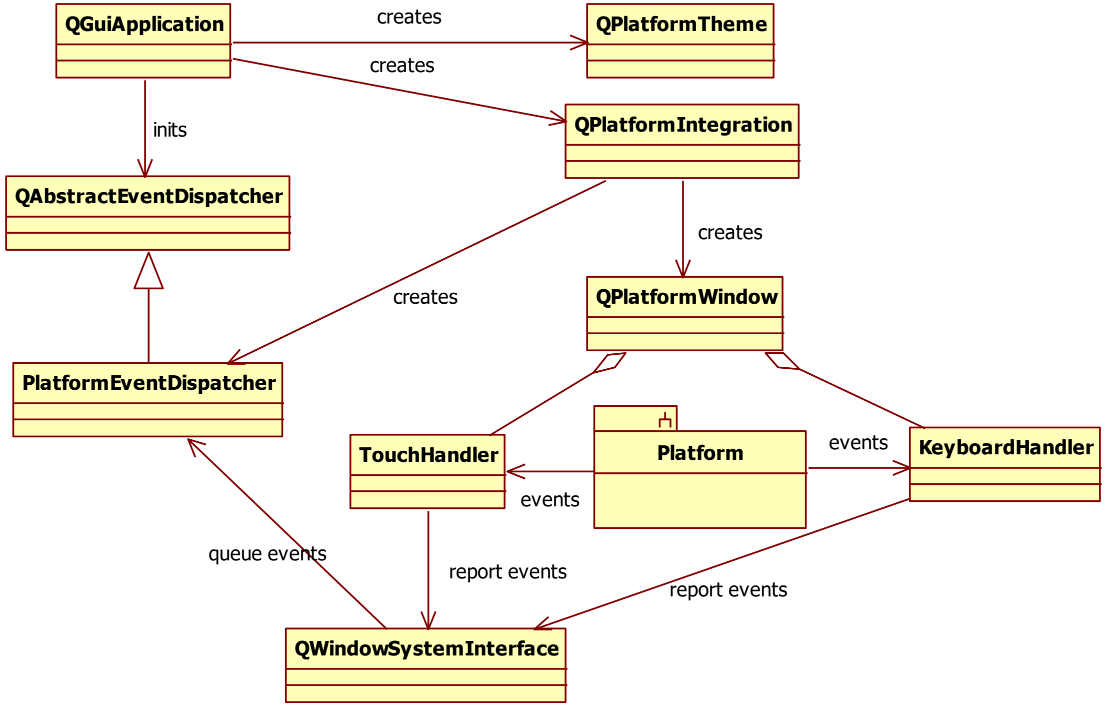
- Supports OpenGL ES and SW rendered windows on top of EGL without a windowing system
- Recommended plugin for embedded Linux with GPU
- Forces the first top-level window to be full screen
  - All other windows (dialogs, popup menus, drop-down windows) are composited to the top-level window
  - EGLFS supports exactly one native window and EGL window surface
  - Opening two OpenGL windows or mixing OpenGL and raster windows is not supported

- If multiple windows are needed, you may use
  - Qt Wayland Compositor plugin

QPA High-Level Architecture

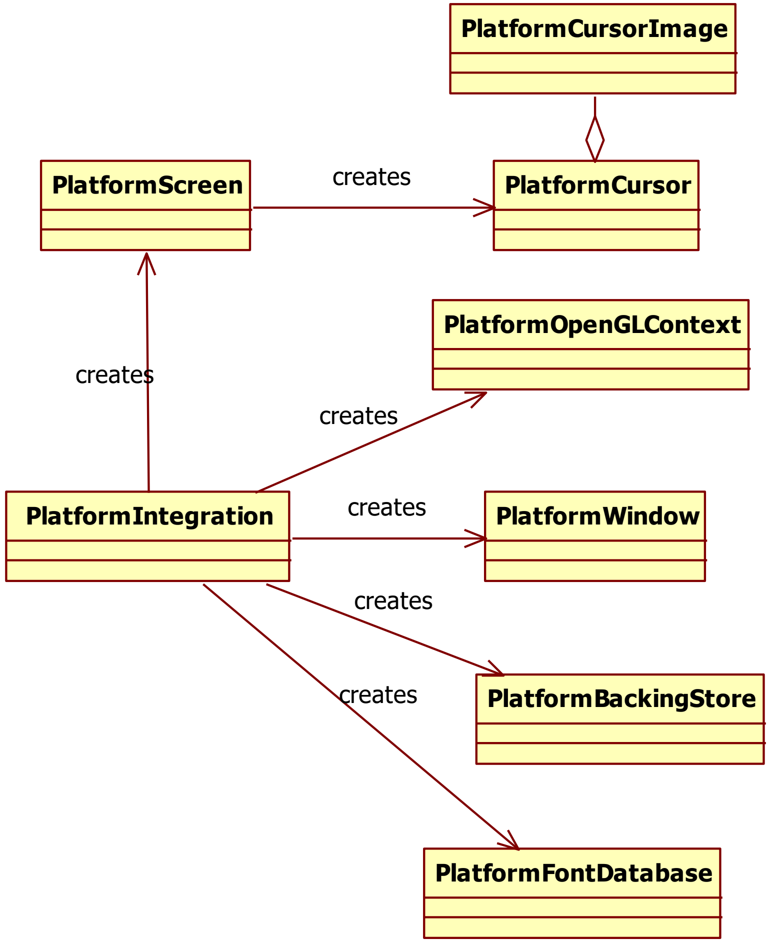
- QPA plugin will be created by `QGuiApplication`
- There are two main classes created
  - `QPlatformTheme` – theming support integration
  - `QPlatformIntegration` – window system integration

- The platform integration class will also create a concrete, platform-dependent event dispatcher for GUI events
  - There exist ready made classes for event dispatching

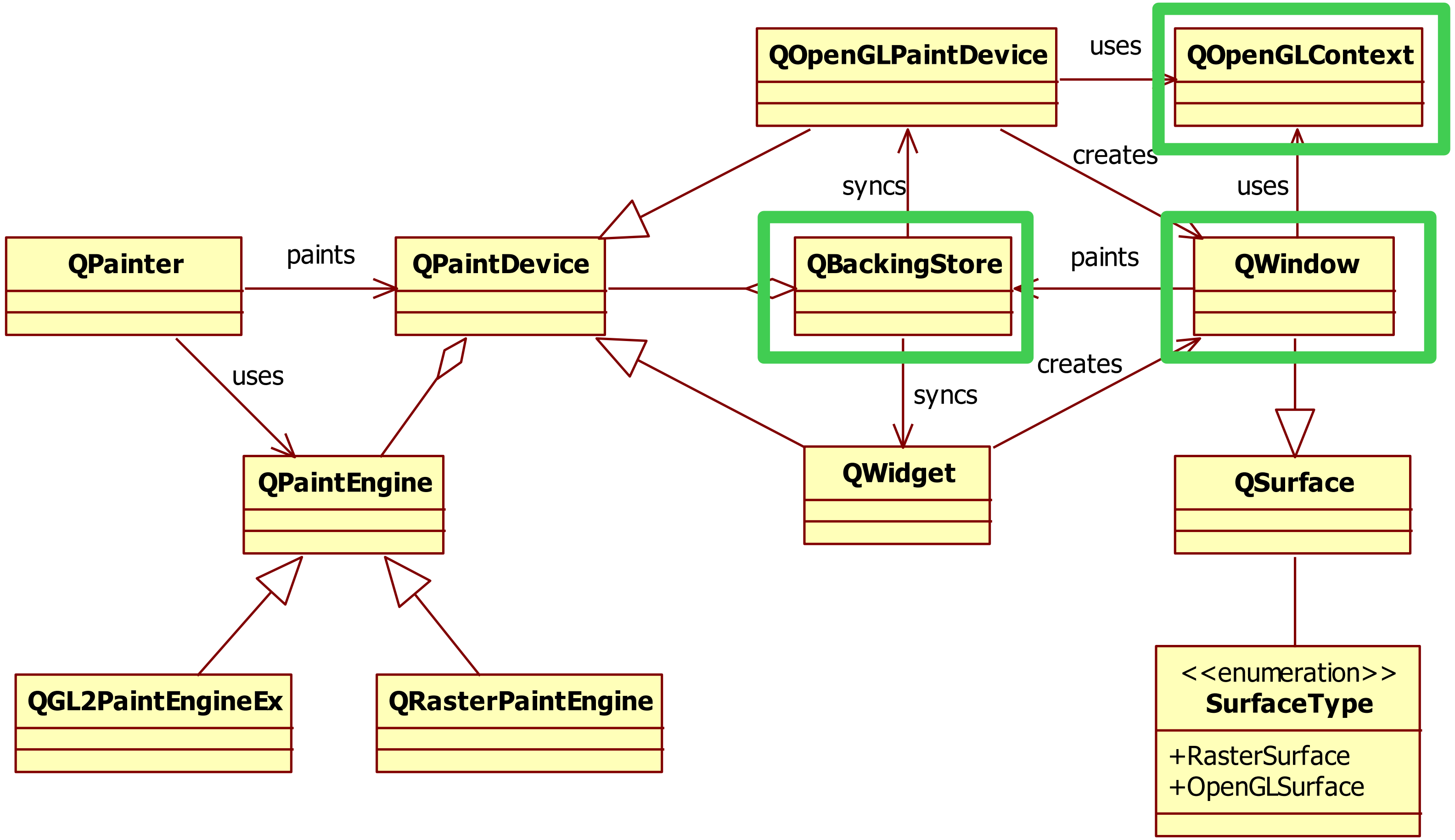


Essential Classes

- In Qt5, it is possible to paint using `QWidget` (a paint device) or `QWindow`
- When the main widget (top-level window) is set visible, a `QWindow` object is created
  - Platform integration class creates a platform window
  - Windows are managed by `QPlatformScreen`, corresponding to `QScreen` class
  - The platform screen often manages GUI event handlers as well
- The window is either a raster or OpenGL surface
  - A raster surface paints to a paint device using the backing store, which flushes the pixels to the frame buffer
  - An OpenGL surface uses a platform dependent `QOpenGLContext`



Paint Classes and Platform Classes



How to Start?

- Select whether you want to have an accelerated QPA plugin or not
  - Acceleration support may be added later
- Trivial example plugins
  - Raster - `qtbase/src/plugins/minimal/`
  - Accelerated - `qtbase/src/plugins/minimalegl/`
- Plenty of ready-made code
  - `qtbase/src/platformsupport/`

- Often it is enough to add or adjust the feature rather than implement a complete plugin from the scratch

Platform Support

- Plenty of useful functions and classes as included projects (**`.pri`**)
- Accessibility
  - Assistive Technology Service Provider Interface + DBus clients
- Basic + font database **`.pri`** projects
  - Font database (accessed through the platform integration class)
- Device discovery
  - Static and udev device manager based device discovery

- EGL (+ GLX)
  - EGL-based GL context
- Input
  - All input devices
- Desktop services
  - `openUrl()`

Essentials Classes – QPlatformIntegration

- Three pure virtual functions
  - `QAbstractEventDispatcher *`*`guiThreadEventDispatcher`*`()`
    - Concrete event dispatchers in the platform support folder
  - `QPlatformWindow *createPlatformWindow(QWindow *window)`
  - `QPlatformBackingStore *createPlatformBackingStore(QWindow *window)`

- Capabilities
  - Threaded pixmaps (re-entrant pixmaps), threaded OpenGL (OpenGL support outside GUI thread), buffer queuing OpenGL (`swapBuffers()` does not immediately suspend the thread), window masks, multiple windows (windows composited)
  - Return true, if the capability is supported

Essentials Classes – QPlatformIntegration

- Constructor
  - Instantiates platform screens
  - Add to the container using `screenAdded()` – can be accessed using `QGuiApplication::screens()`

- Other functions
  - For deeper window integration

Input Handlers

- Platform support **`input.pri`**
  - Mouse, keyboard, touch, tablet
- Can be created anywhere: platform integration, screen, window

- All input handlers share the same principles
  - Read events from the device API using file descriptors
  - You may find QT_OPEN and QT_READ macros useful
  - Use `QSocketNotifie`r to read events asynchronously from the file descriptor
    - `myNotifier = new QSocketNotifier(pointerFD, QSocketNotifier::Read, this);`
    - `QT_READ(pointerFD, bufPos, size);`
  - Implement event data parsing in the notifier callback
  - Use `QWindowSystemInterface` API to add the event to Qt event queue

Example – Input Handlers

```
QEvdevTouchScreenHandler::QEvdevTouchScreenHandler(const QString &specification, QObject *parent) :
    QObject(parent), m_notify(0), m_fd(-1), d(0)
{
    QString dev;
    QScopedPointer<QDeviceDiscovery> deviceDiscovery(QDeviceDiscovery::create(
        QDeviceDiscovery::Device_Touchpad | QDeviceDiscovery::Device_Touchscreen, this));
    if (deviceDiscovery) {
        QStringList devices = deviceDiscovery->scanConnectedDevices();
        dev = devices[0];
    }
    m_fd = QT_OPEN(dev.toLocal8Bit().constData(), O_RDONLY | O_NDELAY, 0);
    if (m_fd >= 0) {
        m_notify = new QSocketNotifier(m_fd, QSocketNotifier::Read, this);
        connect(m_notify, SIGNAL(activated(int)), this, SLOT(readData()));
    }
}

void QEvdevTouchScreenHandler::readData()
{
    int result = QT_READ(m_fd, reinterpret_cast<char*>(buffer) + n, sizeof(buffer) - n);
```

QWindowSystemInterface

- Provides an event queue
  - The event dispatcher will call `sendWindowSystemEvents()` to get the events
- `TouchPoint` structure
  - Similar state to GUI touch points, position, pressure [0, 1]

- Plenty of static functions for
  - Key events
  - Mouse events
  - Window management
  - Drag and drop handling
  - Tablet enter/leave proximity events

Essentials Classes – QPlatformScreen

- Abstraction of physical screens
  - Initializes your monitor
  - Physical size needed to calculate the DPI
  - Three pure virtual functions:
    - `QRect geometry()` – physical dimensions
    - `int depth()` – number of colors
    - `QImage::Format format()` – e.g. `QImage::Format_RGB32`
  - If font point sizes do not map properly to font pixel sizes, implement `QDpi logicalDpi()`
- Also used to manage singleton resources in QPA plug-ins
  - Cursor – platform cursor
  - Mouse, touch, and keyboard drivers (may be created by the platform window as well)
- Container of windows

Implementation Issues

- In some plugins, by default a resolution of 100 dots per inch assumed
  - Re-implement `physicalSize()` and (`logicalDpi()`) for other DPI

```
static const int dpi = 100;
return QSizeF(geometry().size()) / dpi * qreal(25.4);
```

Example QLinuxFbScreen

```
QLinuxFbScreen::QLinuxFbScreen() :
    mFbFd(-1),
    mBlitter(0)
{ }

bool QLinuxFbScreen::initialize(const QStringList &args)
{
    QString fbDevice, ttyDevice;

    if (fbDevice.isEmpty())
        fbDevice = QLatin1String("/dev/fb0");

    mFbFd = openFramebufferDevice(fbDevice);
    mFbScreenImage = QImage(mMmap.data, geometry.width(), geometry.height(), mBytesPerLine,
                            mFormat);
    mCursor = new QFbCursor(this);


    mBlitter = new QPainter(&mFbScreenImage);
```

Essentials Classes – QPlatformWindow

- Describes the window
  - Derives from `QPlatformSurface`, which defines surface type (raster, OpenGL)
  - The content of the window is defined by `QPlatformBackingStore`
- Concrete class
  - Sub-class may use a native window manger –specific window
- The geometry of the top-level widget
  - May need mapping between window manager window geometry
  - `setGeometry()`
- May have child windows
- Window event handling functions
  - From the possible window manager
  - Forwarded to the event system using `QWindowSystemInterface`
- Holds the GL context (if supported)

Implementation Issues

- Mouse grab
- By default the window under the mouse cursor, will receive the event
- If mouse grab is explicitly set to a window, other windows should not receive mouse enter/leave or obviously any other mouse events

Example QFbWindow

- Not sub-classed in LinuxFB QPA plugin
  - Implemented in the platform support
- Calculates dirty regions and repaints after `QFbBackingStore::flush()`

```
void QFbWindow::setGeometry(const QRect &rect)
{
    // store previous geometry for screen update
    mOldGeometry = geometry();
    platformScreen()->invalidateRectCache();

    QWindowSystemInterface::handleGeometryChange(window(), rect);
    QPlatformWindow::setGeometry(rect);
}
```

Essentials Classes – QPlatformBackingStore

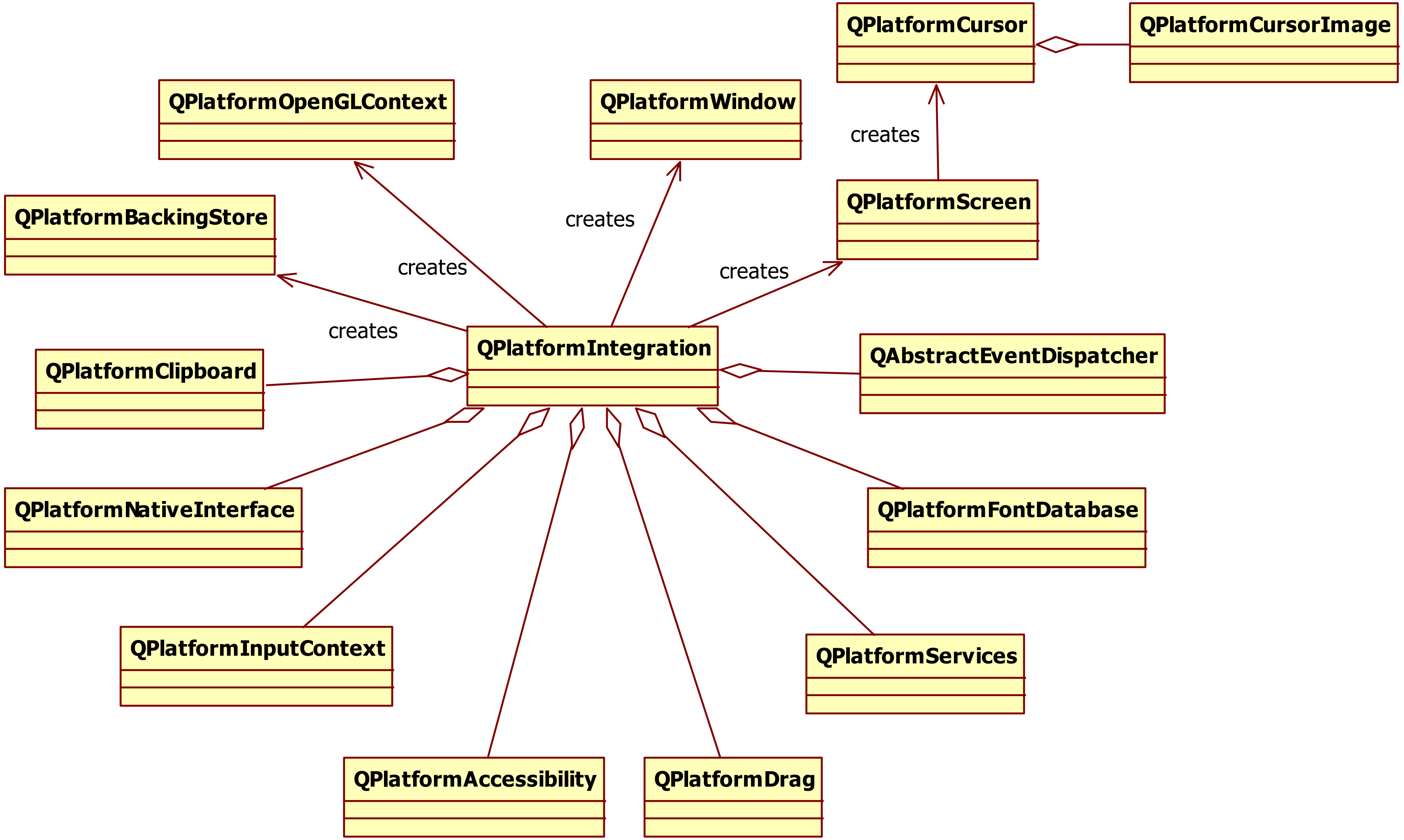
- Describes the content of a top-level window
  - Created when the window created and added to the screen container
  - (Raster) window is rendered to the paint device, specified by the backing store
  - `virtual QPaintDevice *paintDevice() = 0;`
  - E.g. `QImage`, which is mapped to the platform image structure using, e.g. shared memory (LinuxFb)
- Pushes pixels to screen
  - Use the image regions to create the actual image region
  - Push pixels using the platform API
  - The offset parameter indicates a possible image translation with respect to the window
  - `virtual void flush(QWindow *window, const QRegion &region, const QPoint &offset) = 0;`
- Takes care of the window size
  - E.g. create a new platform-specific resized image and a new `QImage` mapped to that
  - `virtual void resize(const QSize &size) = 0;`

Example QFbBackingStore

- Not sub-classed in LinuxFB QPA plugin

```
void QFbBackingStore::flush(QWindow *window, const QRegion &region, const QPoint &offset)
{
    Q_UNUSED(window);
    Q_UNUSED(offset);
    (static_cast<QFbWindow *>(window->handle()))->repaint(region);
}

void QFbBackingStore::resize(const QSize &size, const QRegion &staticContents)
{
    Q_UNUSED(staticContents);
    if (mImage.size() != size)
        mImage = QImage(size, window()->screen()->handle()->format());
}
```

Other Classes



Other Classes

- `QPlatformNativeInterface`
  - Container for native resources (context, window, backing store, screen etc.)
- `QPlatformFontDatabase`
  - Interface to your platform font database
  - Creates `QFontEngine`
  - Existing font databases
    - `QFontconfigDatabase`
    - `QBasicUnixFontDatabase`
    - `QGenericUnixDatabase`
  - Supports FreeType fonts
- `QPlatformServices`
  - Backend for desktop functionality
  - `openURL(), openDocument()`
- `QPlatformCursor`
  - Platform cursor implementation
  - Cursor shapes (arrow, wait), pos
- `QPlatformClipboard`
  - Abstraction of the platform clipboard
  - Copying the data based on MIME types

Other Classes

- `QPlatformDrag`
  - Platform drag abstraction
  - Drag actions: move, copy, link

- `QPlatformInputContext`
  - Interface for implementing input methods
  - When input complex text where simple keymap is not enough

- `QPlatformAccessibility`
  - For integrating accessibility backends

QPlatformTheme

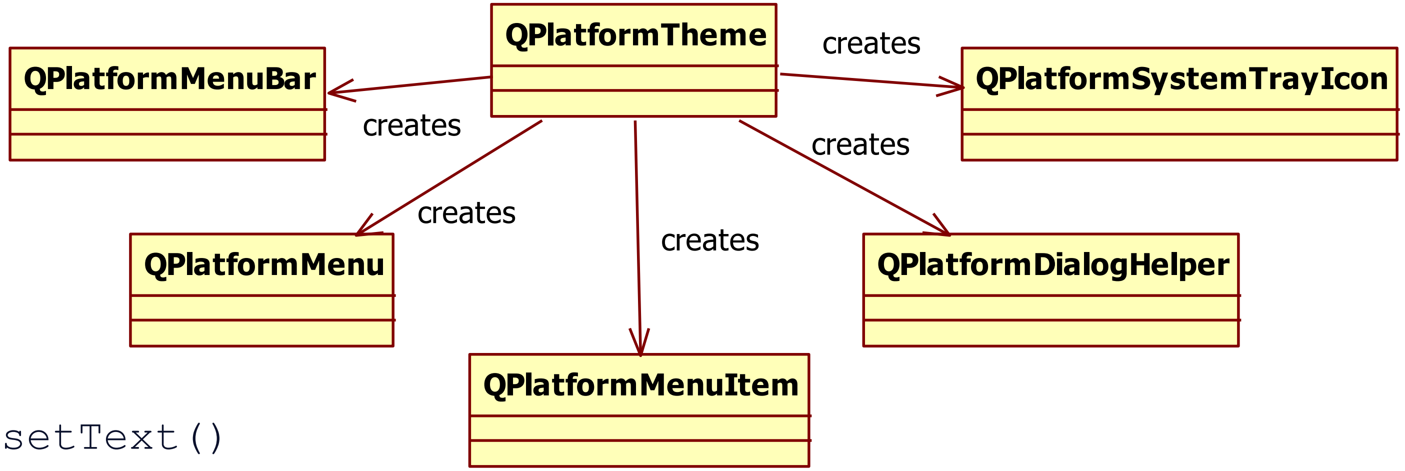
- Theme-based UI customization instead of `QStyle`
- Look at a generic unix theme in **`src/platformsupport/`**`themes`
- Functions and enumerations for the
  - Palette (system palette, button palette, label palette, etc.)
  - Font (system font, menu font, label font, title bar font, etc.)
  - Standard pixmaps (min, max, close buttons, drive icons, directory icons, file icons, etc.)
  - Theme hints (UI effects, icon pixmap sizes, password mask character etc.)

QPlatformTheme

```
QVariant QGtk2Theme::themeHint(QPlatformTheme::ThemeHint hint) const
{
    switch (hint) {
    case QPlatformTheme::SystemIconThemeName:
        return QVariant(gtkSetting("gtk-icon-theme-name"));
    case QPlatformTheme::SystemIconFallbackThemeName:
        return QVariant(gtkSetting("gtk-fallback-icon-theme"));
    default:
        return QGnomeTheme::themeHint(hint);
```

Theming – Menus and Dialogs

- Menus
  - `QPlatformMenuBar`
    - Insert, remove menus
  - `QPlatformMenu`
    - Insert and add items
    - Item at the position
    - `setEnabled(), setVisible(), setText()`
  - `QPlatformMenuItem`
    - Text, icon
- Dialogs
  - `QPlatformDialogHelper`
    - `exec(), show(), hide(), styleHint()`



Platform Themed Dialog

```
QScopedPointer<QGtk2Dialog> d;
QGtk2ColorDialogHelper::QGtk2ColorDialogHelper()
{
    d.reset(new QGtk2Dialog(gtk_color_selection_dialog_new("")));
    connect(d.data(), SIGNAL(accept()), this, SLOT(onAccepted()));
    connect(d.data(), SIGNAL(reject()), this, SIGNAL(reject()));
    GtkWidget *gtkColorSelection = gtk_color_selection_dialog_get_color_selection(
            GTK_COLOR_SELECTION_DIALOG(d->gtkDialog()));
    g_signal_connect_swapped(gtkColorSelection, "color-changed",
            G_CALLBACK(onColorChanged), this);
}
```

Summary

- QPA is a single plugin abstracting the platform
  - Window system
  - Theming
- QPA itself is not a window system, but it may implement a window system
- Roles of the essential QPA classes are
  - Platform integration singleton
  - Platform window corresponding to top-level windows
  - Platform backing store containing the pixels of the top-level raster windows
  - Physical screen abstraction
- Hardware acceleration is much more straightforward compared to Qt Window System in Qt4
  - Create a platform OpenGL context used by QOpenGLContext

Virtual Keyboard

Contents

- Usage
- Customization

Virtual Keyboard

- Available in Windows and Linux desktops and Boot2Qt
  - Implements a platform input context plugin

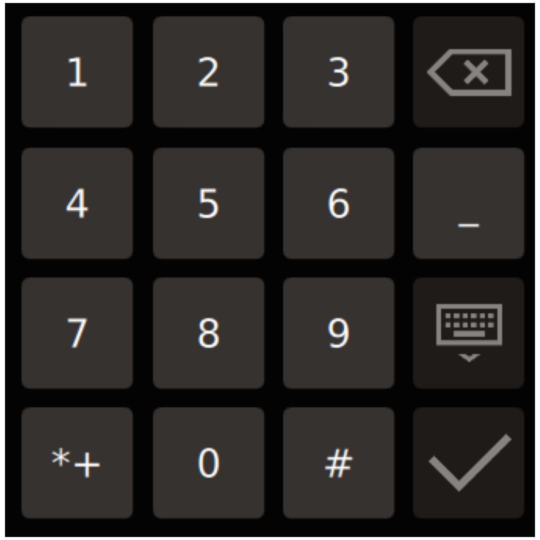
- Basic features
  - Predictive text input, scalable UI, handwriting support with gestures, audio feedback

- Customizable
  - Custom input methods, custom keyboard layouts and styles
  - 3rd party input engines can be integrated

- Localizable
  - support for different character sets, LTR, RTL, dynamic language change

Usage

- Configure and build the VKB from sources
  - `lang-de_DE, lang-all` – use ISO country and language codes
  - `handwriting` – by default enables T9 (if installed), use lipi-toolkit to enable lipi
  - `disable-desktop` – by default desktop VKB enabled on desktop platforms



Two VKB integration methods supported

- Desktop – virtual keyboard is available for all applications without any changes in the apps
- Application – virtual keyboard is available to applications only after the applications have created an instance of QML `InputPanel` element (only available integration method in Boot2Qt)

- An application must load the VKB plugin either using `QT_IM_MODULE` environment variable or
  - Using `qputenv(“QT_IM_MODULE”, QByteArray(“qtvirtualkeyboard”));` in `main()`

Settings – VirtualKeyboardSettings

- Provides simple settings

- Changing the style (retro, default, custom)
  - `VirtualKeyboardSettingslocale.styleName =  “retro”;`
  - `QT_VIRTUALKEYBOARD_STYLE` environment variable may be used as well
  - The custom style is defined in QML using KeyboardLayout

- Changing the locale
  - `VirtualKeyboardSettingslocale.locale =  “fi_FI”;`

Essential Classes – InputPanel

- `InputPanel` – Provides VKB UI
  - Anchor left and right or set width
  - Set `y` coordinate
  - Do not set height, as that is calculated automatically to keep the aspect ratio
  - Define, when visible

```
InputPanel {
    z: 99
    anchors.left: parent.left
    anchors.right: parent.right
    y: parent.height

    visible: Qt.inputMethod.visible rent.right
```

Essential Classes – HandwritingInputPanel

- Provides full screen handwriting input
- Used together with `InputPanel`
  - Hides the input panel and shows handwriting keyboard, when made available (available: true)
  - Activated with active: true
  - UI logic is provided by the developer

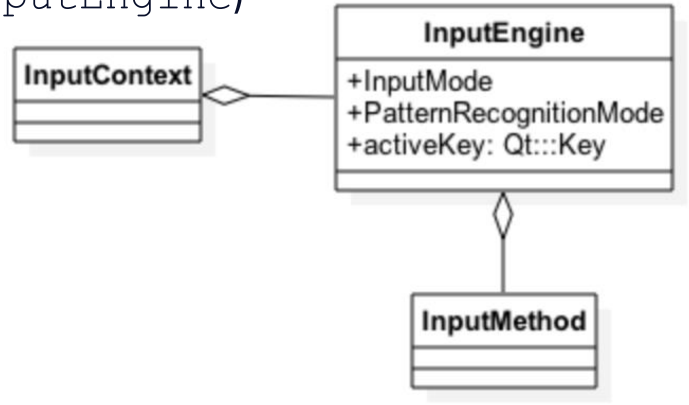
```
HandWritingInputPanel {
    id: hwInputPanel
    anchors.fill: parent
    inputPanel: inputPanelId
    // Decoration to indicate the panel is active

Button {
    anchors.fill: parent
    onClicked: hwInputPanel.active = true;
```

Essential Concepts

- `InputContext` – Provides contextual information for the virtual keyboard and input methods
  - `locale, inputMethodHints (ImhHiddenText, ImhDigitsOnly), pre-edit text, inputItem, cursor position, keyboard rectangle, input engine …)`
  - `if (InputContext.shift === true) { VirtualKeyboardSettings.locale = “fi_FI”; }`

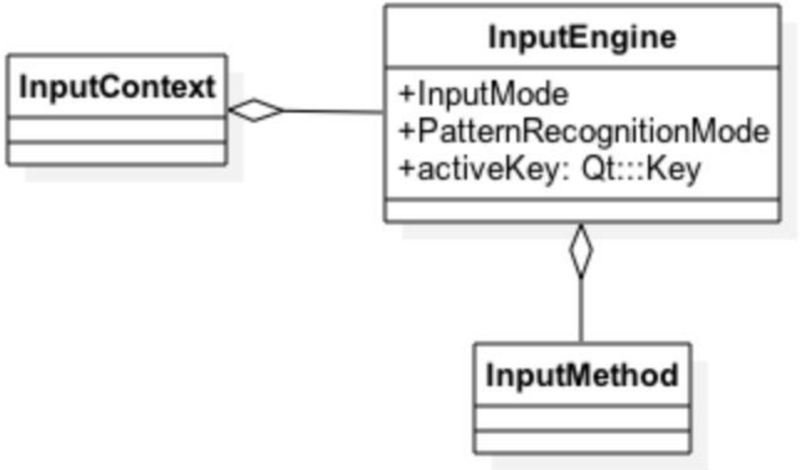
- `InputEngine` – Provides an API to integrate input events
  - `activeKey, virtualKeyPress, virtualKeyRelease, inputMode (latin, numeric, hangul, patternRecognitionMode)`
  - Word selection model
  - Host for input methods (set the input method and it will start receiving keys from `InputEngine`)

- `InputContext` and `InputEngine` are used as singletons



Essential Concepts

- `AbstractInputMethod/InputMethod` – Base class for custom input methods
- Create a custom input method type
  - Instantiate and assign to input engine
  - Set the input mode
- `InputMethod also` provides access to locale-based layout **`virtualkeyboard/layouts/de_DE/symbols.qml`**



Customization – VKB Layouts

- Layouts are provided in a locale-specific folder **`virtualkeyboard/layouts/de_DE/symbols.qml`**
  - Input mode defines the layout type, which is recognized by the file name
  - Some layout types are activated with input method hints (IMH) while other from the main layout
  - main, symbols, numbers (IMH), dialpad (IMH), handwiritng

```
KeyboardLayoutLoader {// Optional for managing several keyboard layouts (pages)
   sourceComponent: Component { KeyboardLayout {

KeyboardLayout {
    inputMethod: handWritingInputMethod // Only in HWR layouts
    keyWeight: 200 // Decoration to indicate the panel is active
    KeyboardRow {
        Key { key: Qt.Key_A; text: “a”; alternativeKey: [ “ä”, “å” ] ]
        BackspaceKey { weight: 400 ]
        HandwritingModeKey { noModifier: true ]
        TraceInputKey { patternRecognitionMode: InputEngine.HandwirtingRecognition ]
        // Collects and renders touch input data
```

Customization – Styles

- Defined in the **`virtualkeyboard/content/styles`** folder
  - Copy a default style folder
  - Replace images
  - Change the `style.qml` file
  - Set the resource prefix according to the location of your actual resources
    - Like “”, if in the same folder
- Some stylable properties
  - Fonts
  - Icons
  - Icon scale
  - Background color
  - Margins
  - Key panel, containing a sound effect and control

Summary

- Qt provides a cross-platform virtual keyboard for Linux, Boot2Qt and Windows
- VKB can be used in the application and desktop mode
  - Application mode means `InputPanel` is created in code
  - Desktop mode is automatic
- VKB supports easy localization and customization

Serial Bus

Contents

- Serial Bus Usage
- Backends

Qt SerialBus

- Abstracts industrial serial buses and protocols

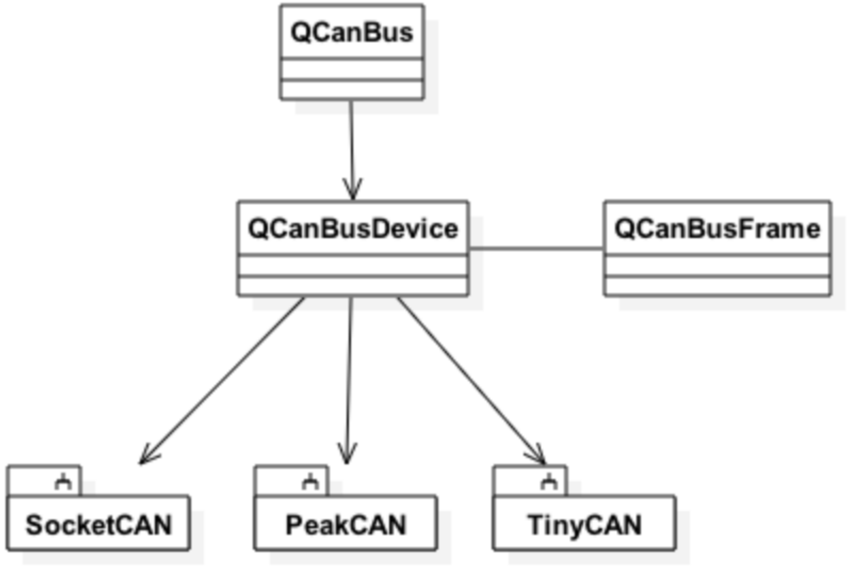
- Controller Area Network (CAN)
  - Used mainly by automotive but also in cycling, industrial, and entertainment applications
  - Multi-master serial bus standard for connecting sensors and other control units
  - Units send and receive messages AKA frames to each other

- ModBus
  - Request/reply protocol with one master and several slaves
  - Master reads and writes data into the slaves
  - Supports any `QIODevice`, provided the plugin implements `QIODevice` functions

- Implemented in its own module
  - `QT += serialbus`

Serial Bus Usage – CAN Bus

- The CAN bus API provides a common API + vendor-specific plugin
- Two classes in the common API
  - `QCanBusDevice` – direct access to the CAN device
    - Reads and writes messages using the backend
  - `QCanBusFrame` – defines a message, which can be read or written to the CAN device
    - Provides an identifier, payload, and a timestamp, when the frame was read
- Check the backend exists
- Create the device
- Set the configuration, if needed (bit rate (not supported in the CAN bus), frame filters etc.)
- Read and write frames
  - Signal `QCanBusDevice::framesReceived()` emitted, when new frames received



CAN Bus Usage Example

```
Q_FOREACH (const QByteArray &backend, QCanBus::instance()->plugins()) {
    if (backend == "socketcan") { // or “peakcan” or “tinycan”
        // Plugin was found
        break; } }

// CAN interfaces, like can0, can be requested using ifconfig
// Peak can supports only USB adapters usbbus1 to usbbus8
// Tiny-CAN supports only two interfaces: channela and channelb
QCanBusDevice *device = QCanBus::createDevice("socketcan", QStringLiteral("can0"));
// Some configuration params, like bit rate, must be set before the connection
device->connectDevice();

QCanBusFrame frame;
frame.setFrameId(8);
QByteArray payload("A36E");
frame.setPayload(payload);
device->writeFrame(frame);

QCanBusFrame frame = device->readFrame();
```

CAN Bus Backends

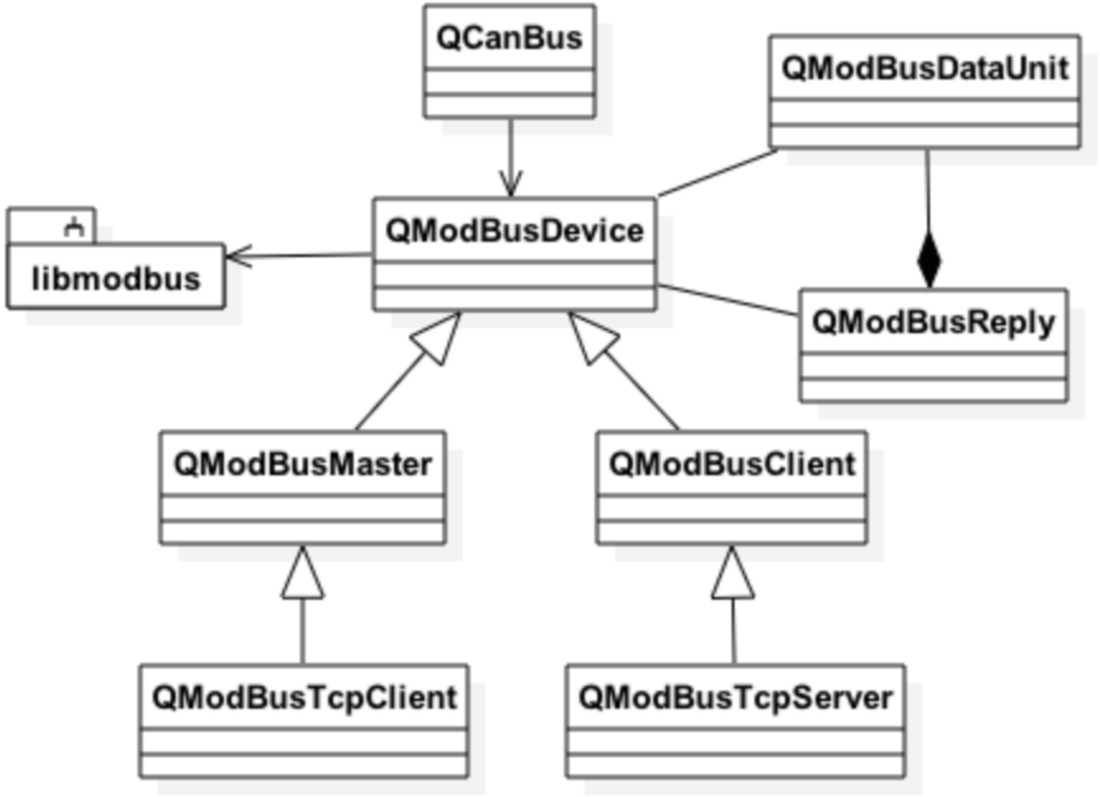
- Vendor-specific APIs are implemented in the backend plugin

- Three backends supported
  - SocketCAN using Linux sockets and open source drivers
  - PeakCAN – using PCAN adapters
  - TinyCAN – using Tiny-CAN adapters

- Use `QCanBus` to register and create bus backends
  - Create the CAN bus device: `QCanBus::createDevice(const QByteArray &plugin, const Qstring &interfaceName)`
  - Get a pointer to `QCanBus` singleton: `static QCanBus::instance()`
  - Return a list of loaded plugin identifiers: `QCanBus::plugins();`

Serial Bus Usage – ModBus

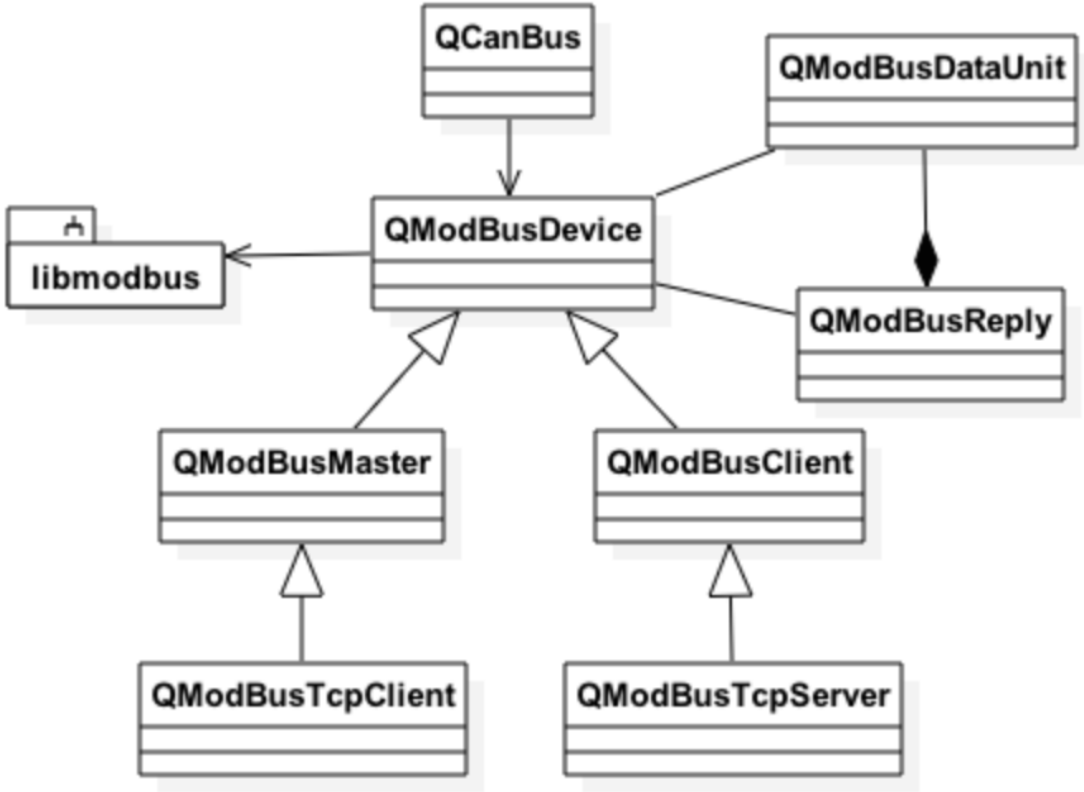
- QModBusDevice – QModBusMaster and QModBusSlave base class
- QModBusMaster – Communicates with the backend using read()/write() functions
  - Subclass QModBusTcpClient
- QModBusSlave – Direct access to ModBus slave
  - Subclass QModBusTcpServer



Serial Bus Usage – ModBus

- `QModBusDataUnit` – Container, representing single bit or 16-bit entries in ModBus registers
  - Discrete Input and Coils are single bit register types
  - Several registers may be access from the start address and the number of contiguous entries
  - Input registers are read only as holding registers may be read and written

- `QModBusReply` – Contains the data and address for the request
  - Signal finished() emitted after a request is successfully completed



ModBus Usage Example

```
Q_FOREACH (const QByteArray &backend, QCanBus::instance()->plugins()) {
    if (backend == "libmodbus") {
        // ModBus backend found
        break;
    }
}

// Create a master or slave, depending on the device
QModBusMaster *master = QModBus::createMaster("libmodbus");

// Initialize slave tables
modBusSlave->setMap(QModBusDevice::DiscreteInputs, 10);
modBusSlave->setMap(QModBusDevice::Coils, 10);
modBusSlave->setMap(QModBusDevice::InputRegisters, 10);
modBusSlave->setMap(QModBusDevice::HoldingRegisters, 10);
```

ModBus Usage Example

```
// Set a connection to the network and connect the device
// TCP uses QModBusDevice::TCP and serial port QModBusDevice::RemoteTerminalUnit
// package type, respectively
QSerialPort *serialPort = new QSerialPort("ttyS0");
modBusSlave->setDevice(serialPort, QModBusDevice::RemoteTerminalUnit);
modBusSlave->setSlaveId(1);
modBusSlave->connectDevice();

// Read or write a single or multiple data units
units.append(QModBusDataUnit(QModBusDevice::HoldingRegisters, 3, 0x1af5));
units.append(QModBusDataUnit(QModBusDevice::HoldingRegisters, 4, 0x1001));
units.append(QModBusDataUnit(QModBusDevice::HoldingRegisters, 5, 0xff34));
modBusMaster->write(units);
```

ModBus Backends

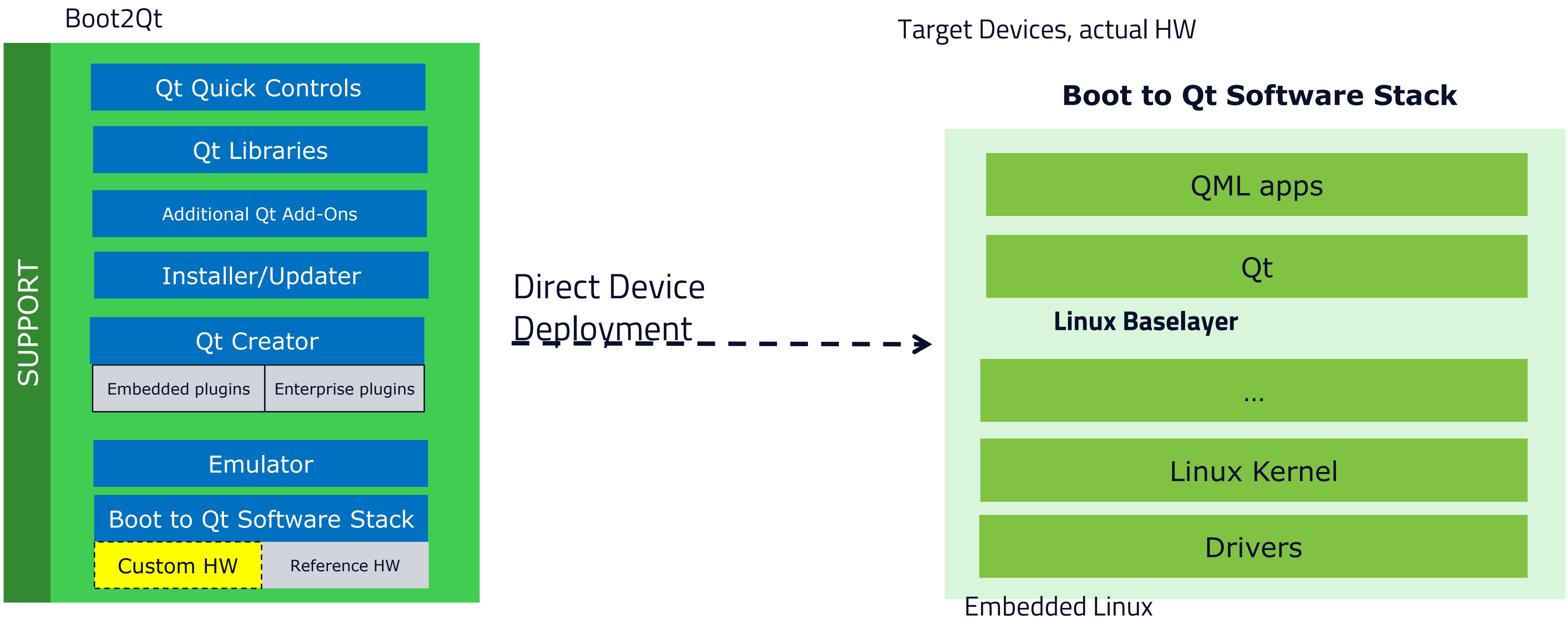
- One free SW backend library, libmodbus supported
- Supports serial and Ethernet communication

Boot2Qt

Contents

- Boot2Qt
- Embedded App Creation, Building, Debugging, and Deployment
- Build System Customization

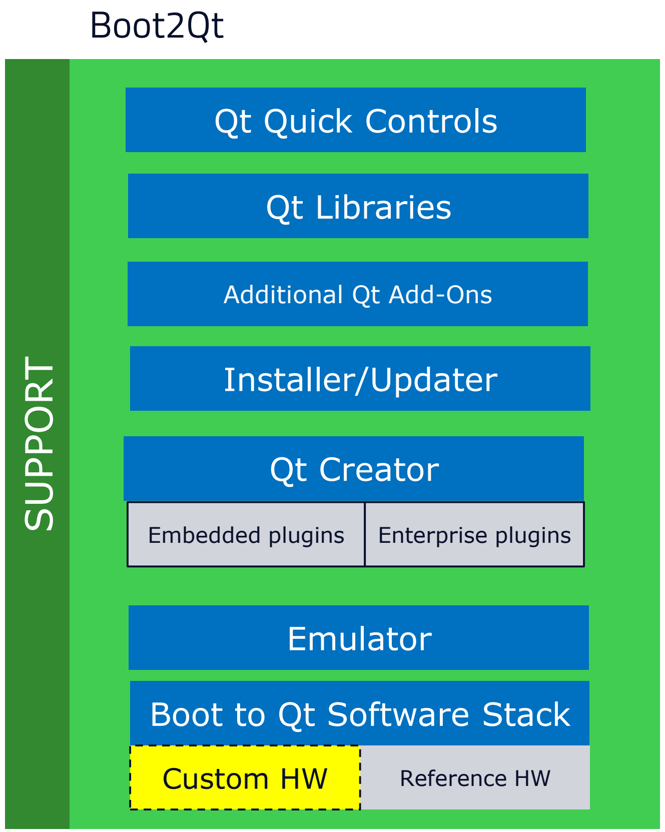
Boot2Qt - Contents



Boot2Qt - Value

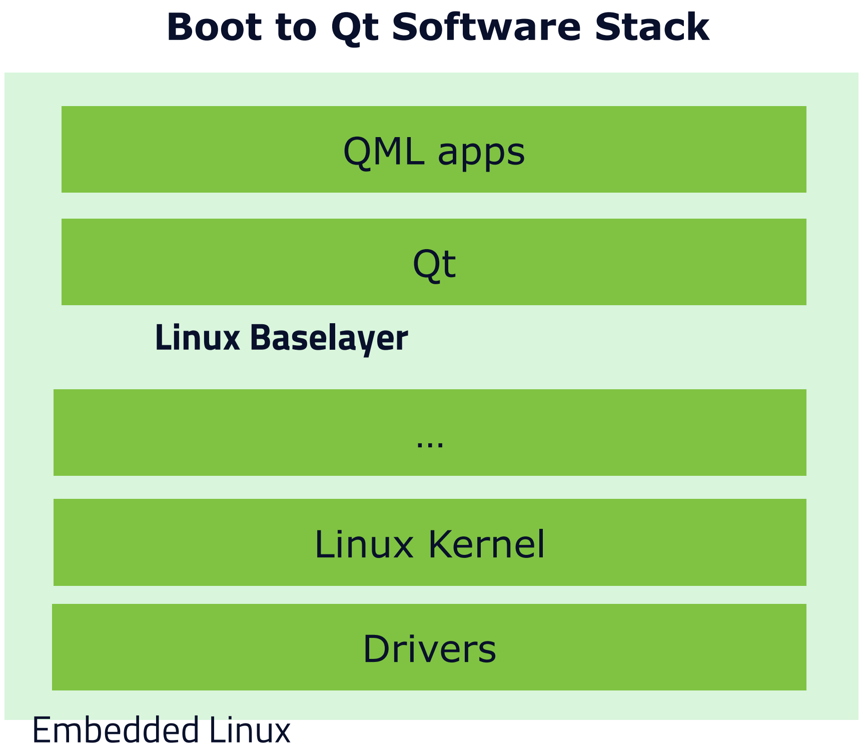
- Out-of-the-box device creation with Qt
  - Embedded Device creation has never been this easy!
- Professional convenience and cost-effective tooling around Qt libraries
  - *Run it on the device in just hours!*

- *Boot to Qt Software Stack*
  - Pre-built, lightweight, Qt-optimized software stack for embedded Linux
  - Custom HW support through The Qt Company



Boot to Qt Software Stack – Embedded Linux

- The Embedded Linux variant provides exactly the same software stack than for Android but with different kernel
- The Embedded Linux stack is built using Yocto recipes for Poky system
  - Boot to Qt Software Stack for Embedded Linux is *Yocto compliant*
- Provides greater customization possibilities for the stack if one wants to replace parts of Boot to Qt
  - for instance custom WLAN component, etc.
- The Qt Company helps in customization of the stack!



Supported Platforms and Toolchains

- Raspberry Pi, Raspberry Pi 2
- BeagleBone Black
- Freescale SABRE SD i.MX6Dual, iMX6Quad
- Boundary Devices i.MX6 Boards (QNX)
- Toradex Apalis and Colibri i.MX6, Colibri VF
- SILICA ArchiTech Tibidabo
- Emulator

- Any toolchain used for Linux building may be downloaded and used
  - Boot2Qt uses Yocto Poky reference system version 1.6

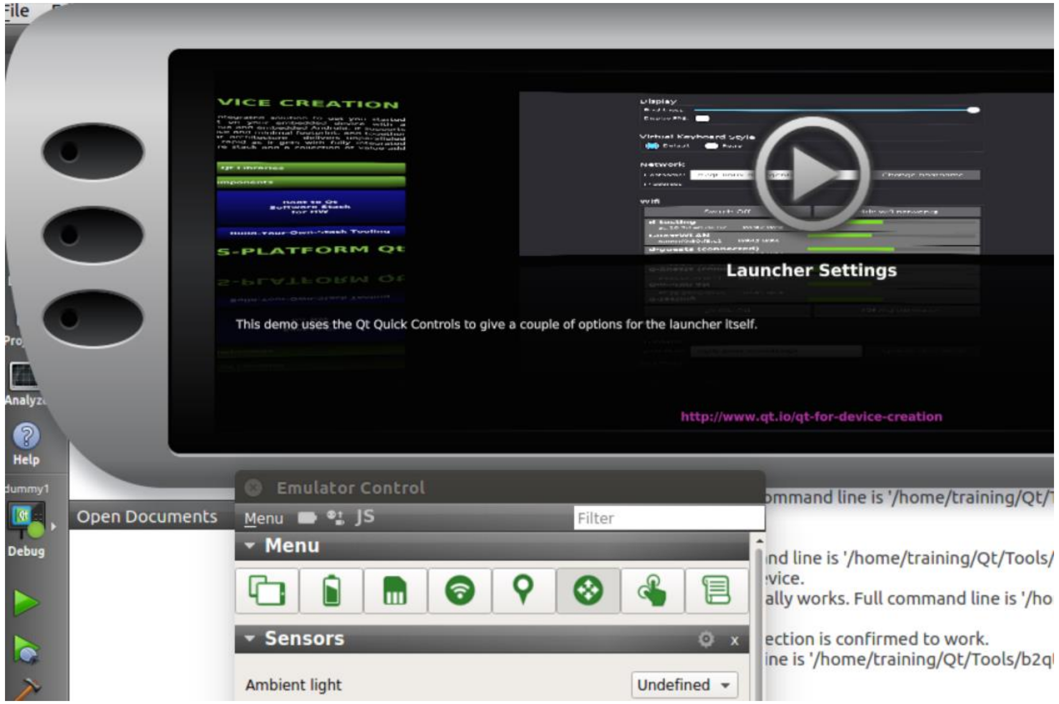
How to Get Started?

- Pre-built target images and toolchains
- Can be customized and built from the sources as well
- Install Linux image to the SD card
  - `sudo <Boot2Qt>/5.5/Boot2Qt/<device>-eLinux/images/deploy.sh /dev/<device>`
- Connect to the target using either Ethernet or USB
  - Android **`adb`** tool is used for device connection (Android Debug Bridge)
  - Ethernet
  - USB – the user account must have access to the plugged in devices
    - `echo ‘SUBSYSTEM==“usb”, ATTRS{idVendor}==“18d1”, TAG+=“udev-acl”, TAG+=“uaccess”’ | sudo tee –a /etc/udev/rules.d/70-boot2qt.rules`
  - Check the connection
    - `<Boot2Qt>/Tools/b2qt/adb devices -l`

Emulator

- Useful to run programs without the HW but with a similar configuration

- Device model (dashboard, tablet, DPI)
- Battery capacity, level, flow, voltage
- SD storage
- WiFi connection
- Location (latitude, longitude, altitude, direction, speed)
- Sensors(ambient light, orientation, compass, proximity)
- Multipoint touches
- Scripts



This demo uses the Qt Quick Controls to give a couple of options for the launcher itself.

http://www.qt.io/qt-for-device-creation

```
ommand line is '/home/training/Qt/T

nd line is '/home/training/Qt/Tools/
vice.
ally works. Full command line is '/ho

ection is confirmed to work.
ine is '/home/training/Qt/Tools/b2q
```

Creating Custom Builds

- Install Qt Enterprise Embedded source packages using Qt binary installer
- Install dependencies: `gawk, wget, got-core, diffstat, unzip, p7zip-full, txinfo, gcc-multilib, build-essential, chrpath, libsdl1.2-dev, xterm, gperf, bison, curl, udisks, screen`
- Init Yocto
  - In the build folder, call

    ```
    <Boot2Qt>/5.5/Boot2Qt/sources/b2qt-yocto-meta/b2qt-init-build-env init –device <device>
    ```
  - You may `use list-devices` option to see all the devices
- Configure the build environment
  - `export MACHINE=<machine>`
  - `source ./setup-environment.sh`
- Build the targets (Qt Enterprise Embedded contains two targets)
  - `bitbake b2qt-embedded-image`
  - `bitbake meta-toolchain-b2qt-embedded-sdk`
  - Note! No Qt libraries built yet

Creating Custom Builds

- Build Qt libraries
  - Setup the build environment

    `<Boot2Qt>/5.5/Boot2Qt/sources/b2qt-build-scripts/embedded-common/init_build_env.sh`

    `<Boot2Qt>/5.5/Boot2Qt/sources/b2qt-build-scripts/embedded/embedded-linux/config.<machine>`
  - Build the libraries
    - Qt libs - `./build_qt.sh`
  - Demos, add-ons, Qt WebEngine - `./build_extras.sh`
  - Creates an image containing Qt libs in the rootfs **`/usr/local/`** and the complete image to be deployed to the target - `./build_image.sh`
- Copy the image to the SD card
  - `sudo ./deploy.sh /dev/<dev_name>`

- Add a new kit to QtCreator
  - `<Boot2Qt>/5.5/Boot2Qt/sources/b2qt-build-scripts/embedded-common/setup_qtcreator.sh`

Summary

- Boot to Qt provides pre-built binaries for many embedded targets
  - Possible to concentrate on app development starting from day 1
- Possible to configure and build the root file system and Qt libraries for custom platforms as well
- Emulator allows SW testing without the actual HW
  - Can be configured to have similar features to target HW
- Deployment and on-device debugging are supported by QtCreator

Thank you