Qt & CMake

State of the Union *(revised)*

@alcroito

A bit of history

- Initial work on porting Qt to CMake started ~ September 2018
  - Parts of qtbase ported
  - Implemented initial scripts for automatic conversion
  - Was done in a custom repo on git.qt.io

- Merged to gerrit/wip/cmake at end of October 2018
  - Previous history became one single squashed commit
  - Built on Windows, macOS, Linux
  - rasterwindow example rendered on screen :)
  - No Coin : (

Current Status

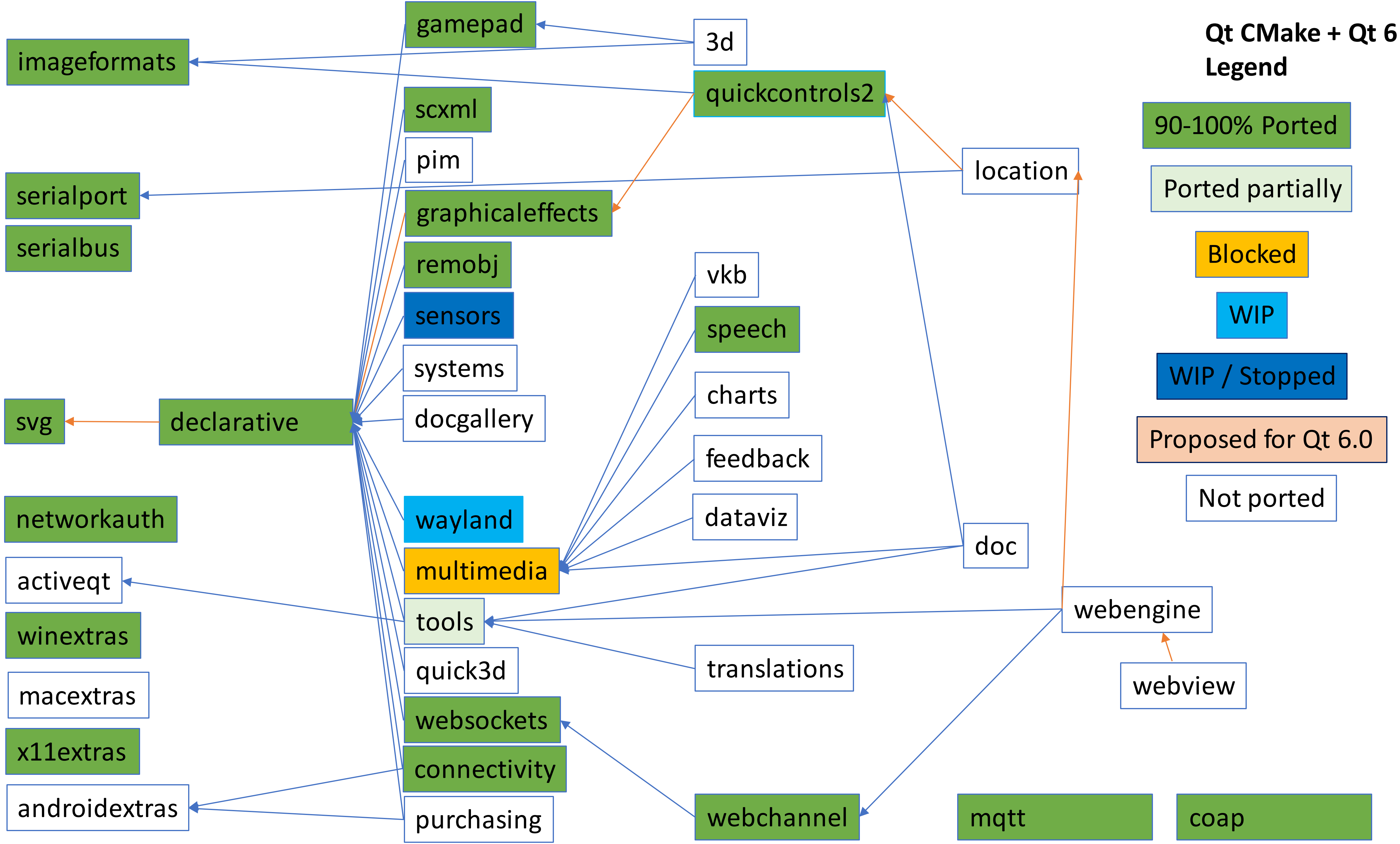
- Many repos build!
- Many examples build!
- Many tests build!
- Works on more platforms!
- Coin tests our changes!

Built repos *(revised)*

- qtbase
- qtsvg
- qtimageformats
- qtgraphicaleffects
- qtdeclarative
- qtquickcontrols2
- qtnetworkauth
- qtmqtt
- qttools (qdoc only)
- qtcoap
- qtconnectivity
- qtgamepad
- qtknx
- qtremoteobjects
- qtscxml
- qtserialbus
- qtserialport
- qtwebsockets
- qtwebchannel
- qtwinextras
- qtx11extras
- qtwayland (almost)

Missing repos

- qtlocation
- qt3d
- qtwebengine
- qtmultimedia
- qtcharts
- qtvirtualkeyboard
- ….


Qt CMake + Qt 6
Legend
90-100% Ported
Ported partially
Blocked
WIP
WIP / Stopped
Proposed for Qt 6.0
Not ported
gamepad
3d
imageformats
quickcontrols2
scxml
pim
location
serialport
serialbus
graphicaleffects
remobj
sensors
systems
docgallery
svg
declarative
vkb
speech
charts
feedback
dataviz
networkauth
wayland
multimedia
doc
activeqt
tools
webengine
winextras
quick3d
translations
webview
macextras
websockets
x11extras
connectivity
androidextras
purchasing
webchannel
mqtt
coap

Working platforms *(revised)*

- Windows Desktop
- Linux Desktop
- macOS
- Static builds
- Android

Tested in Coin

- iOS (works, WIP)
- MinGW (works, been a while)
- Embedded Linux (works, been a while)
- WebAssembly (works, been a while)

- Didn't try WinRT

Things left to do

- Remaining repos, examples, tests
- Certain more isoteric parts of qtbase
- Documentation
- Translations
- Super builds (qt5)
- debug_and_release
- qmake mixing (don’t ask)
- QtWebEngine
- Polishing
- …
- Lots of other stuff I forgot

Future plan

- Merge wip/cmake to dev soon ™
  - There are some blockers due to new moc + json functionality in dev
- Build Qt with qmake and CMake at the same time (while transitioning)
- …
- Profit

How to build Qt

- How to build qtbase
- https://github.com/qt/qtbase/blob/wip/cmake/cmake/README.md

- Porting tips and tricks guide
- https://wiki.qt.io/CMake_Port/Porting_Guide

- Upstream CMake documentation
- https://cmake.org/cmake/help/v3.15/

- #qt-cmake on Freenode IRC

Requirements

- CMake version 3.15.0+ @ https://cmake.org/download/
- Checkout of qtbase wip/cmake branch git://code.qt.io/qt/qtbase.git
- Ninja @ https://ninja-build.org/
- Your favourite compiler (Clang, gcc, Apple clang, MSVC)
- A source for dependencies
  - vcpkg @ https://github.com/microsoft/vcpkg (can be used for Windows, macOS, Linux)
  - Homebrew for macOS
  - apt-get, dnf, zypper, emerge, pacman, yum, <insert-favourite-tool-here> for Linux

- Additional requirements for non-desktop platforms (not mentioned here)
- Some good will

A little bit of CMake

CMake crash course interlude

What you should know about CMake:

- It‘s syntax is horrifying
- It has a bunch of concepts:
  - variables, cache variables, functions, macros, commands, targets, target properties, generator expressions
- It has a configuration step and a generation step
- It generates Makefiles, ninja files, Xcode projects, MSVC projects, etc.

Simple CMake app

```
cmake_minimum_required(VERSION 3.11.0)
project(myapp LANGUAGES CXX)

set(sources "main.cpp")
add_executable(myapp ${sources})

target_compile_definitions(myapp PRIVATE "-DMY_AWESOME_DEFINE=1")
target_compile_options(myapp PRIVATE "-g")
target_link_options(myapp PRIVATE "-Wl,--gc-sections")
```

A bit of CMake syntax

```
function(my_func value)
  message("Value is: ${value}")
  set(my_var "${value}" PARENT_SCOPE)
endfunction()

macro(my_macro value)
  message("Value is: ${value}")
  set(my_var "${value}")
endmacro()
```

Debugging CMake configuration step

```
$ cmake ../qtbase -trace &> log.txt …
```

Or

```
$ cmake ../qtbase -trace-expand &> log.txt …
```

Or with CMake 3.16+

```
$ cmake ../qtbase –trace-redirect=log.txt …
```

Similar to "`qmake -d`", "`qmake -d -d`".

Building Qt

How we build Qt with qmake (ok, how I build it)

```
$ mkdir qt60_built && cd qt60_built && mkdir qtbase && cd qtbase
$ /path/to/qt60_source/qtbase/configure -developer-build …
$ make -j16

$ cd .. && mkdir qtdeclarative && cd qtdeclarative
$ /path/to/qt60_built/qtbase/bin/qmake /path/to/qt60_source/qtdeclarative
$ make -j16
```

How to build Qt with CMake

```
$ mkdir qt60_built && cd qt60_built && mkdir qtbase && cd qtbase
$ cmake /path/to/qt60_source/qtbase -DFEATURE_developer_build=ON -GNinja
$ ninja

$ cd .. && mkdir qtdeclarative && cd qtdeclarative
$ /path/to/qt60_built/qtbase/bin/qt-cmake /path/to/qt60_source/qtdeclarative -GNinja
$ ninja
```

How to build just widgets

```
$ cd qt60_built/qtbase/src/widgets
$ make -j8

$ cd qt60_built/qtbase
$ ninja Widgets
```

How to build all tests in a subfolder

```
$ cd qt60_built/qtbase/tests/auto/widgets
$ make -j8

$ cd qt60_built/qtbase
$ ninja tests/auto/widgets/all
```

How to run tests with qmake and CMake

```
$ make check
```

```
$ ninja test
```

or

```
$ cd build_dir && ctest -V -R tst_my_test_name
```

-V – Verbose

-R – Regex to match test names

A little bit of Python

Conversion scripts

- qtbase/util/cmake/pro2cmake.py
- qtbase/util/cmake/configurejson2cmake.py
- qtbase/util/cmake/run_pro2cmake.py
- qtbase/util/cmake/helper.py (helper)

pro2cmake.py

- Main script for conversion
- Takes a .pro file, spits out a CMakeLists.txt file
- Uses Python 3.7
- Output is not always perfect, might need manual adjustments
  - Is much improved since a few months ago
  - Unfortunately, won't work correctly when .pro files are not declarative enough
- How to install dependencies:
  - `$ python3.7 -m pip install -r qtbase/util/cmake/requirements.txt`
- Example usage:
  - `python3 ./pro2cmake.py /path/to/qtbase/src/gui/gui.pro`
  - Will create a CMakeLists.txt file in src/gui

configurejson2cmake.py

- Main script for converting configure.json files
- Takes a folder containing configure.json, spits out a configure.cmake file
- Also uses Python 3.7
- No manual adjustments in the generated file are allowed (verboten!)
- Example usage:
  - `$ python3 ./configurejson2cmake.py /path/to/qtbase/src/gui/`
  - Will create a configure.cmake file in src/gui

run_pro2cmake.py

- Useful script when you want to convert many projects at once
  - E.g. examples, tests
- Finds all the .pro files that it thinks need to be converted (not always 100% correct)
  - Is recursive
- Example usage:
  - `$ python3 ./run_pro2cmake.py /path/to/qtbase/tests/auto`
- Various useful options:
  - `--only-existing`
  - `--skip-subdirs-projects`
  - `--only-missing`
  - `--count 10`
  - `--offset 20`

# special case + pro2cmake.py

- Until the port is finished, there is a need to merge from dev -> wip/cmake
  - Changes in .pro / .qrc / .json files
  - Manual syncing of .pro and CMakelists.txt files is a pain
  - Certain things need to be manually handled because pro2cmake is not good enough

- Rerunning pro2cmake.py will try to regenerate only the parts that changed
- But it requires some user input
- Manually modified changes in CMakeLists.txt need to be annotated with
  - either **`# special case`**
  - or **`# special case begin`** and **`# special case end`** comment blocks

How does preservation work?

- The special case preservation mechanism uses a .prev_CMakeLists.txt file
- Needs to be committed together with other changes
  - Automatically git add'ed when running pro2cmake
  - Also generated for the first time when special cases are found
    - Which means no .prev file for projects that have no modifications
- Uses git under the hood

Let’s look at some Qt CMake details

Sample repo project in qmake

```
$ cat qtsvg.pro

load(qt_parts)
```

Sample repo project in CMake

```
$ cat qtsvg/CMakeLists.txt

cmake_minimum_required(VERSION 3.15.0)
project(QtSvg VERSION 6.0.0 DESCRIPTION "Qt SVG Libraries" HOMEPAGE_URL "https://qt.io/"
    LANGUAGES CXX C
)

find_package(Qt6 ${PROJECT_VERSION} CONFIG REQUIRED COMPONENTS BuildInternals Core Gui
Widgets)
find_package(Qt6 ${PROJECT_VERSION} CONFIG OPTIONAL_COMPONENTS Xml) # For tests
qt_build_repo() # <-- same as load(qt_parts)
```

Sample src project in qmake

```
$ cat qtsvg/src/src.pro

TEMPLATE = subdirs
CONFIG += ordered
qtHaveModule(gui): SUBDIRS += svg plugins
```

Sample src project in CMake

```
$ cat qtsvg/src/CMakeLists.txt

add_subdirectory(svg)
add_subdirectory(plugins)
```

Sample module project in qmake

```
$ cat qtsvg/src/svg/svg.pro

TARGET      = QtSvg
QT          = core-private gui-private
HEADERS += qsvggraphics_p.h …
SOURCES += qsvggraphics.cpp …

qtConfig(system-zlib): QMAKE_USE_PRIVATE += zlib
qtHaveModule(widgets): QT += widgets-private
```

Sample module project in CMake

```
$ cat qtsvg/src/svg/CMakeLists.txt

find_package(ZLIB MODULE REQUIRED) # special case
qt_add_module(Svg
    SOURCES qgraphicssvgitem.cpp qgraphicssvgitem.h …
    DEFINES QT_NO_USING_NAMESPACE
    LIBRARIES Qt::CorePrivate Qt::GuiPrivate ZLIB::ZLIB
    PUBLIC_LIBRARIES Qt::Core Qt::Gui)
qt_extend_target(Svg
   CONDITION TARGET Qt::Widgets
   LIBRARIES  Qt::WidgetsPrivate PUBLIC_LIBRARIES Qt::Widgets)
```

Sample plugin project in qmake

```
$ cat qtsvg/src/plugins/imageformats/svg/svg.pro

TARGET  = qsvg
HEADERS += qsvgiohandler.h
SOURCES += main.cpp qsvgiohandler.cpp
QT += svg

PLUGIN_TYPE = imageformats
PLUGIN_EXTENDS = svg
PLUGIN_CLASS_NAME = QSvgPlugin
load(qt_plugin)
```

Sample plugin project in CMake

```
$ cat qtsvg/src/plugins/imageformats/svg/CMakeLists.txt

qt_add_plugin(qsvg
    TYPE imageformats
    CLASS_NAME QSvgPlugin
    SOURCES main.cpp qsvgiohandler.cpp qsvgiohandler.h
    PUBLIC_LIBRARIES
        Qt::Core
        Qt::Gui
        Qt::Svg
)
```

Sample configure.json

```
$ cat qtimageformats/src/imageformats/configure.json
```

```
"libraries": {
  "jasper": {
    "label": "jasper",
    "test": {
      "label": "Jasper(header in /usr/include)",
      "type": "compile",
      "test": {
        "include": [
          "string.h",
          "jasper/jasper.h"
        ],
        "qmake": [
          "msvc: LIBS += libjasper.lib",
          "else: LIBS += -ljasper"
        ],
        "main": [
          "// This version of Jasper is broken, according to the old Qt Solutions docs",
          "if (strcmp(JAS_VERSION, \"1.900.0\") == 0)",
          "    return 1;",
          "return 0;"
        ]
      }
    },
    "sources": [
      "-ljasper"
    ]
  },
```

```
  "mng": {
    "label": "mng",
    "test": {
      "label": "MNG(header in /usr/include)",
      "type": "compile",
      "test": {
        "include": [
          "stdio.h",
          "libmng.h"
        ],
        "qmake": [
          "LIBS += -lmng"
        ],
        "main": [
          "mng_handle hMNG;",
          "mng_cleanup(&hMNG);",
        "#if defined(MNG_VERSION_MAJOR)",
        "#if MNG_VERSION_MAJOR < 1 || (MNG_VERSION_MAJOR == 1 &&
MNG_VERSION_MINOR == 0 && MNG_VERSION_RELEASE < 9)",
        "#error System libmng version is less than 1.0.9",
        "#endif",
        "#endif",
          "return 0;"
        ]
      }
    },
    "sources": [
      "-lmng"
    ]
  },
```

```
"features": {
  "jasper": {
    "label": "JasPer",
    "disable": "input.jasper == 'no'",
    "condition": "features.imageformatplugin && libs.jasper",
    "output":[
      "privateFeature",
      { "type": "define", "negative": true, "name": "QT_NO_IMAGEFORMAT_JASPER" }
    ]
  },
  "mng": {
    "label": "MNG",
    "disable": "input.mng == 'no'",
    "condition": "libs.mng",
    "output":[ "privateFeature" ]
  },
  "tiff": {
    "label": "TIFF",
    "disable": "input.tiff == 'no'",
    "condition": "features.imageformatplugin",
    "output":[
      "privateFeature"
    ]
  },
```

```
  "system-tiff": {
    "label": " Using system libtiff",
    "disable": "input.tiff == 'qt'",
    "enable": "input.tiff == 'system'",
    "condition": "features.tiff && libs.tiff",
    "output":[ "privateFeature" ]
  },
  "webp": {
    "label": "WEBP",
    "disable": "input.webp == 'no'",
    "condition": "features.imageformatplugin",
    "output":[
      "privateFeature"
    ]
  },
  "system-webp": {
    "label": " Using system libwebp",
    "disable": "input.webp == 'qt'",
    "enable": "input.webp == 'system'",
    "condition": "features.webp && libs.webp",
    "output":[ "privateFeature" ]
  }
},
```

Sample configure.cmake

```
$ cat qtimageformats/src/imageformats/configure.cmake

#### Libraries

qt_find_package(WrapJasper PROVIDED_TARGETS WrapJasper::WrapJasper)
qt_find_package(TIFF PROVIDED_TARGETS TIFF::TIFF)
qt_find_package(WrapWebP PROVIDED_TARGETS WrapWebP::WrapWebP)
```

FindWrapJasper.cmake

```
$ cat qtimageformats/cmake/FindWrapJasper.cmake

set(WrapJasper_FOUND OFF)
find_package(Jasper)
if(Jasper_FOUND)
    set(WrapJasper_FOUND ON)
    # Upstream package does not provide targets, only variables. So define a target.
    add_library(WrapJasper::WrapJasper INTERFACE IMPORTED)
    target_link_libraries(WrapJasper::WrapJasper INTERFACE ${JASPER_LIBRARIES})
    target_include_directories(WrapJasper::WrapJasper INTERFACE ${JASPER_INCLUDE_DIR})
endif()
```

Sample configure.cmake continued

```
$ cat qtimageformats/src/imageformats/configure.cmake

#### Features

qt_feature("jasper" PRIVATE
    LABEL "JasPer"
    CONDITION QT_FEATURE_imageformatplugin AND WrapJasper_FOUND
    DISABLE INPUT_jasper STREQUAL 'no'
)
qt_feature_definition("jasper" "QT_NO_IMAGEFORMAT_JASPER" NEGATE)
```

Jasper image format plugin in CMake

```
$ cat qtimageformats/src/plugins/imageformats/jp2/CMakeLists.txt

qt_add_plugin(qjp2
    TYPE imageformats
    CLASS_NAME QJp2Plugin
    SOURCES main.cpp qjp2handler.cpp qjp2handler_p.h
    LIBRARIES
        WrapJasper::WrapJasper
        Qt::Gui # special case
)
```

Command line stuff

Mapping of qmake features to CMake features

- Take the feature name in configure json, replace dashes with underscores, prepend FEATURE_
  - `-feature-clock-monotonic -> -DFEATURE_clock_monotonic=ON`
  - `-no-feature-itemmodel -> -DFEATURE_itemmodel=OFF`

- `ON  – value for enabling feature (has to be all upper case)`
- `OFF – value for disabling feature (has to be all upper case)`

In the future we should massage configure to do the transformation for us. (WIP)

Command line arguments you might want to use

- `-DFEATURE_developer_build=ON` – enables private tests and no need to ninja install
- `-DQT_USE_CCACHE=ON` - enable ccache for faster recompiling (Linux and macOS only)
- `-DCMAKE_INSTALL_PREFIX=/path/to/qtbase_installed` - for specifying install prefix
- `-DCMAKE_BUILD_TYPE=Debug` (or `Release`)
- `-DBUILD_EXAMPLES=OFF`
- `-DBUILD_TESTING=ON`
- `-DQT_NO_MAKE_EXAMPLES=ON` - Similar to -nomake examples
- `-DQT_NO_MAKE_TESTS=ON` - Similar to -nomake tests
- `-GNinja` (or `–GUnix Makefiles`)

CMakeCache.txt

- Stores computed features and passed command line arguments
  - Similar to .qmake.cache or .qmake.stash
- You can edit it manually to flip some variables (like `-DBUILD_TESTING=OFF`)
  - Or you can use ccmake (CLI GUI) or cmake-gui (GUI GUI)

- If something is wrong with your build, instead of rm-ing the build folder, try rm-ing just CMakeCache.txt and reconfigure with cmake

Qt Creator

- By the way you can use Qt Creator for building and navigating CMake projects :)
- For best results use Qt Creator 4.11+ and CMake 3.15+

Coin

Current state:

› Staging is disabled for wip/cmake branches.

› Instead, a bot automatically starts a build with each pushed patch set.

Future plan:

› If we merge wip/cmake to dev, Coin will build with both qmake and CMake

› For qmake, old hardcoded build / test instructions are used

› For CMake, we use .yaml files to specify build and test instructions

Thanks. Questions?