Qt for macOS

macOS (previously known as OS X or Mac OS X) is Apple's operating system for the Mac line of computers. It's a UNIX platform, based on the Darwin kernel, and behaves largely similar to other UNIX-like platforms. The main difference is that X11 is not used as the windowing system. Instead, macOS uses its own native windowing system that is accessible through the Cocoa API.

To download and install Qt for macOS, follow the instructions on the Getting Started with Qt page. To build Qt from source, see Qt for macOS - Building from Source.

Supported Versions

When talking about version support on macOS, it's important to distinguish between the build environment; the platform you're building on or with, and the target platforms; the platforms you are building for. The following macOS versions are supported.

Target Platform	Architecture	Build Environment	
macOS 10.14, 10.15, 11, 12	x86_64, x86_64h, and arm64	Xcode 12 (11 SDK),	Xcode 13 (12 SDK)

Build Environment

The build environment on macOS is defined *entirely* by the Xcode version used to build your application. Xcode contains both a toolchain (compiler, linker, and other tools), and a macOS platform-SDK (headers and libraries). Together these define how your application is built.

> **Note:** The version of macOS that you are *running* Xcode on does not matter. As long as Apple ships a given Xcode version that runs on your operating system, the build environment will be defined by that Xcode version.

Xcode can be downloaded from Apple's developer website (including older versions of Xcode).

You should always be using the latest Xcode available from Apple that has been tested with the Qt version you are using. By always building against the latest available platform SDK, you ensure that Qt can take advantage of new features introduced in recent versions of macOS.

Once installed, choosing an Xcode installation is done using the `xcode-select` tool.

```
$ sudo xcode-select --switch /Applications/Xcode.app
```

You can inspect the globally selected Xcode installation using the same tool.

```
$ xcode-select -print-path
/Applications/Xcode.app/Contents/Developer
```

The `xcrun` command can then be used to find a particular tool in the toolchain.

```
$ xcrun -sdk macosx -find clang
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/clang
```

or show the platform SDK path used when building.

```
$ xcrun -sdk macosx --show-sdk-path
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.14.sdk
```

Target Platforms

Building for macOS utilizes a technique called *weak linking* that allows you to build your application against the headers and libraries of the latest platform SDK, while still allowing your application to be deployed to macOS versions lower than the SDK version. When the binary is run on a macOS version lower than the SDK it was built with, Qt will check at runtime whether or not a platform feature is available before utilizing it.

In theory this would allow running your application on every single macOS version released, but for practical (and technical) reasons there is a lower limit to this range, known as the *deployment target* of your application. If the binary is launched on a macOS version below the deployment target macOS or Qt will give an error message and the application will not run.

Qt expresses the deployment target via the `CMAKE_OSX_DEPLOYMENT_TARGET` or `QMAKE_MACOSX_DEPLOYMENT_TARGET` variables, which has a default value set by Qt. You only need to change this default if you know that your own code uses macOS APIs that are only available in a particular version higher than what Qt defaults to. For example, with CMake:

```
set(CMAKE_OSX_DEPLOYMENT_TARGET "10.15")
```

or with qmake:

```
QMAKE_MACOSX_DEPLOYMENT_TARGET = 10.15
```

> **Note:** You should not lower the deployment target beyond the default value set by Qt. Doing so will likely lead to crashes at runtime if the binary is then deployed to a macOS version lower than what Qt expected to run on.

For more information about SDK-based development on macOS, see Apple's developer documentation.

Opting out of macOS behavior changes

One caveat to using the latest Xcode version and SDK to build your application is that macOS's system frameworks will sometimes decide whether or not to enable behavior changes based on the SDK you built your application with.

For example, when dark-mode was introduced in macOS 10.14 Mojave, macOS would only treat applications built against the 10.14 SDK as supporting dark-mode, and would leave applications built against earlier SDKs with the default light mode look. This technique allows Apple to ensure that binaries built long before the new SDK and operating system was released will still continue to run without regressions on new macOS releases.

Building against an older SDK is a last-resort solution, and should only be applied if your application has no other ways of working around the problem.

Architectures

By default, Qt will build for the architecture of your development machine - either `x86_64`, or `arm64` if you are on an Apple Silicon Mac.

To build for other architectures you can use the `CMAKE_OSX_ARCHITECTURES` and `QMAKE_APPLE_DEVICE_ARCHS` variables in your project files or on the command line. This allows you to both cross-compile to a different architecture, and to build universal (multi-architecture) binaries. For example, to build your application for both `x86_64` and `arm64` with CMake:

```
cmake ~/src/myapp -DCMAKE_OSX_ARCHITECTURES="x86_64;arm64"
```

or with qmake:

```
qmake ~/src/myapp QMAKE_APPLE_DEVICE_ARCHS="x86_64 arm64"
```

When specifying the architectures in a project file they should not be quoted, e.g.:

```
TEMPLATE = app
SOURCES = main.cpp
QMAKE_APPLE_DEVICE_ARCHS = x86_64 arm64
```

Generating Xcode Project Files

By default, CMake and qmake generates project files in Makefile format. If you prefer to build and debug your application from within Xcode, you can request that an Xcode project is generated instead:

```
cmake ~/src/myapp -GXcode
```

or with qmake:

```
qmake ~/src/myapp -spec macx-xcode
```

Deploying Applications on macOS

macOS applications are typically deployed as self-contained application bundles. The application bundle contains the application executable as well as dependencies such as the Qt libraries, plugins, translations and other resources you may need. Third party libraries like Qt are normally not installed system-wide; each application provides its own copy.

To build your application as an application bundle with CMake, set the `MACOSX_BUNDLE` property on your executable target. With qmake, bundles are the default. Set `CONFIG -= app_bundle` in your project file (`.pro`) to disable it.

A common way to distribute applications is to provide a compressed disk image (.dmg file) that the user can mount in Finder. The deployment tool, `macdeployqt` (available from the macOS installers), can be used to create the self-contained bundles, and optionally also create a .dmg archive.

Applications can also be distributed through the Mac App Store. macdeployqt (bin/macdeployqt) can be used as a starting point for app store deployment. To ensure that Qt complies with the app store sandbox rules, Qt must be configured with the `-feature-appstore-compliant` argument.

For details about deployment on macOS, see Qt for macOS - Deployment.

> **Note:** For selling applications in the macOS App Store, special rules apply. In order to pass validation, the application must verify the existence of a valid receipt before executing any code. Since this is a copy protection mechanism, steps should be taken to avoid common patterns and obfuscate the code that validates the receipt as much as possible. Thus, this cannot be automated by Qt, but requires some platform-specific code written specifically for the application itself. More information can be found in Apple's documentation.

macOS Issues

The page below covers specific issues and recommendations for creating macOS applications.

- Qt for macOS - Specific Issues

Where to Go from Here

We invite you to explore the rest of Qt. We prepared overviews to help you decide which APIs to use and our examples demonstrate how to use our API.

- Qt Overviews - list of topics about application development
- Examples and Tutorials - code samples and tutorials
- Qt Reference Pages - a listing of C++ and QML APIs

Qt's vibrant and active community site, http://qt.io houses a wiki, a forum, and additional learning guides and presentations.