Variables

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.

Platform-specific variables follow the naming pattern of the variables which they extend or modify, but include the name of the relevant platform in their name. For example, a makespec may use `QMAKE_LIBS` to specify a list of libraries that each project needs to link against, and `QMAKE_LIBS_X11` would be used to extend this list.

ANDROID_ABIS

> **Note:** This variable applies only to Android targets.

Specifies a list of Android target ABIs. Valid values are: armeabi-v7a, arm64-v8a, x86, x86_64.

You can provide the ABIs as a qmake argument:

```
qmake ANDROID_ABIS="armeabi-v7a arm64-v8a"
```

> **Note:** It is possible to use this variable inside the *.pro file, however, it is not recommended since it will override any ABIs specified on the `qmake` command line.

ANDROID_API_VERSION

> **Note:** This variable applies only to Android targets.

Specifies the Android API level number. For more information, see Android Build Numbers.

ANDROID_APPLICATION_ARGUMENTS

> **Note:** This variable applies only to Android targets.

"android.app.arguments". This takes a string of arguments:

```
ANDROID_APPLICATION_ARGUMENTS = "arg1 arg2 arg3"
```

ANDROID_BUNDLED_JAR_DEPENDENCIES

> **Note:** This variable applies only to Android modules.

This is useful when writing a Qt module. It specifies a list of pre-bundled dependencies used by the module in a `.jar` format, for example:

```
ANDROID_BUNDLED_JAR_DEPENDENCIES += jar/Qt6Android.jar
```

ANDROID_DEPLOYMENT_DEPENDENCIES

> **Note:** This variable applies only to Android targets.

By default, androiddeployqt will detect the dependencies of your application. However, since run-time usage of plugins cannot be detected, there could be false positives, as your application might depend on any plugin that is a potential dependency. If you want to minimize the size of your APK, it's possible to override the automatic detection using the this variable. This should contain a list of all Qt files which need to be included, with paths relative to the Qt install root.

> **Note:** Only the Qt files specified with this variable are included. Failing to include all the correct files can result in crashes. It's also important to make sure the files are listed in the correct loading order. This variable provides a way to override the automatic detection entirely, so if a library is listed before its dependencies, it will fail to load on some devices.

ANDROID_DEPLOYMENT_SETTINGS_FILE

> **Note:** This variable applies only to Android targets.

Specifies the path to the `android-deployment-settings.json` file needed by androiddeployqt and `androidtestrunner`. This overrides the path to the settings file generated by qmake, thus you have to make sure to provide a valid settings file.

ANDROID_EXTRA_LIBS

A list of external libraries that will be copied into your application's `libs` folder and loaded on start-up. This can be used, for instance, to enable OpenSSL in your application. For more information, see Adding OpenSSL Support for Android.

To include external libraries for multiple ABIs, where each ABIs has its own directory, use the following:

```
for (abi, ANDROID_ABIS): ANDROID_EXTRA_LIBS += $$PWD/$${abi}/library_name.so
```

Otherwise, if the ABI is included in the library name, use the following:

```
for (abi, ANDROID_ABIS): ANDROID_EXTRA_LIBS += $$PWD/library_name_$${abi}.so
```

ANDROID_EXTRA_PLUGINS

> **Note:** This variable applies only to Android targets.

Specifies a path to C++ plugins or resources that your application has to bundle but that cannot be delivered through the assets system, such as QML plugins. With this variable, androiddeployqt will make sure everything is packaged and deployed properly.

`ANDROID_EXTRA_PLUGINS` must point to the directory where the extra plugin(s) are built. In addition, the build directory structure must follow a naming convention similar to Qt plugins, that is, *plugins/<plugin name>*.

The plugins libraries should have the name format *libplugins_<type>_<name>_<abi>.so*. To achieve that the plugin pro file could be defined as follows:

```
TEMPLATE = lib
CONFIG += plugin

PLUGIN_TYPE = imageformats
DESTDIR = $$top_builddir/plugins/myplugin
TARGET = $$qt5LibraryTarget(myplugin, "plugins/$$PLUGIN_TYPE/")
```

with top_builddir defined in .qmake.conf as:

```
top_builddir=$$shadowed($$PWD)
```

This will ensure that the correct name mangling is applied to the plugin library (*plugins/myplugin/libplugins_imageformats_myplugin_armeabi-v7a.so*).

Then, assuming an extra image format plugin *myplugin* is built as *$$DESTDIR/plugins/myplugin/*, the following ensures it is packaged correctly:

ANDROID_FEATURES

Note: This variable applies only to Android modules.

Specifies a module's features list:

```
ANDROID_FEATURES += android.hardware.location.gps
```

For more information, see Android <uses-feature> Docs.

ANDROID_LIB_DEPENDENCIES

Note: This variable applies only to Android modules.

This is useful when writing a Qt module. It specifies a list of pre-built dependencies used by the module, for example:

```
ANDROID_LIB_DEPENDENCIES += \
    plugins/libplugins_platforms_qtforandroid.so
```

ANDROID_MIN_SDK_VERSION

Note: This variable applies only to Android targets.

Specifies the minimum Android API level for the project. By default, this variable is set to API level 23.

ANDROID_PACKAGE_SOURCE_DIR

Note: This variable applies only to Android targets.

Specifies the path for a custom Android package template. The Android package template contains:

- AndroidManifest.xml file
- build.gradle file and other Gradle scripts

androiddeployqt tool copies the application template from the Qt for Android installation path into your project's build directory, then it copies the contents of the path specified by this variable on top of that, overwriting any existing files. For instance, you can make a custom `AndroidManifest.xml` for your application, then place this directly into the directory specified by this variable.

ANDROID_PERMISSIONS

> **Note:** This variable applies only to Android modules.

Specifies a module's permissions list:

```
ANDROID_PERMISSIONS += android.permission.ACCESS_FINE_LOCATION
```

For more information, see Android <uses-permission> Docs.

ANDROID_TARGET_SDK_VERSION

> **Note:** This variable applies only to Android targets.

Specifies the target Android API level for the project. By default, this variable is set to API level 30.

ANDROID_VERSION_CODE

> **Note:** This variable applies only to Android targets.

Specifies the application's version number. For more information, see Android App Versioning.

ANDROID_VERSION_NAME

> **Note:** This variable applies only to Android targets.

Specifies the application's version in as a human readable string. For more information, see Android App Versioning.

CONFIG

Specifies project configuration and compiler options. The values are recognized internally by qmake and have special meaning.

The following CONFIG values control compiler and linker flags:

Option	Description
release	The project is to be built in release mode. If `debug` is also specified, the last one takes effect.
debug	The project is to be built in debug mode.
debug_and_release	The project is prepared to be built in *both* debug and release modes.
debug_and_release_target	This option is set by default. If `debug_and_release` is also set, the debug and release builds end up in separate debug and release directories.
build_all	If `debug_and_release` is specified, the project is built in both debug and release modes by default.
autogen_precompile_source	Automatically generates a `.cpp` file that includes the precompiled header file specified in the .pro file.
ordered	When using the `subdirs` template, this option specifies that the directories listed should be processed in the order in which they are given. **Note:** The use of this option is discouraged. Specify dependencies as described in the SUBDIRS variable documentation.
precompile_header	Enables support for the use of precompiled headers in projects.
precompile_header_c (MSVC only)	Enables support for the use of precompiled headers for C files.
warn_on	The compiler should output as many warnings as possible. If `warn_off` is also specified, the last one takes effect.
warn_off	The compiler should output as few warnings as possible.
exceptions	Exception support is enabled. Set by default.
exceptions_off	Exception support is disabled.
ltcg	Link time code generation is enabled. This option is off by default.
rtti	RTTI support is enabled. By default, the compiler default is used.
rtti_off	RTTI support is disabled. By default, the compiler default is used.
stl	STL support is enabled. By default, the compiler default is used.
stl_off	STL support is disabled. By default, the compiler default is used.
thread	Thread support is enabled. This is enabled when CONFIG includes `qt`, which is the default.
no_utf8_source	Specifies that the project's source files does not use the UTF-8 encoding. Instead, the compiler default is used.
hide_symbols	Set the default visibility of symbols in the binary to hidden. By default, the compiler default is used.
c99	C99 support is enabled. This option has no effect if the compiler does not support C99, or can't select the C standard. By default, the compiler default is used.
Option	Description

c17	C17, also known as C18, support is enabled. This option has no effect if the compiler does not support C17, or can't select the C standard. By default, the compiler default is used.
c18	This is an alias for the `c17` value.
strict_c	Disables support for C compiler extensions. By default, they are enabled.
c++11	C++11 support is enabled. This option has no effect if the compiler does not support C++11, or can't select the C++ standard. By default, support is enabled.
c++14	C++14 support is enabled. This option has no effect if the compiler does not support C++14, or can't select the C++ standard. By default, support is enabled.
c++17	C++17 support is enabled. This option has no effect if the compiler does not support C++17, or can't select the C++ standard. By default, support is enabled.
c++1z	Obsolete alias for c++17.
c++20	C++20 support is enabled. This option has no effect if the compiler does not support C++20, or can't select the C++ standard. By default, support is disabled.
c++2a	Obsolete alias for c++20.
c++latest	Support for the latest C++ language standard is enabled that is supported by the compiler. By default, this option is disabled.
strict_c++	Disables support for C++ compiler extensions. By default, they are enabled.
depend_includepath	Appending the value of INCLUDEPATH to DEPENDPATH is enabled. Set by default.
lrelease	Run `lrelease` for all files listed in TRANSLATIONS and EXTRA_TRANSLATIONS. If `embed_translations` is not set, install the generated .qm files into QM_FILES_INSTALL_PATH. Use QMAKE_LRELEASE_FLAGS to add options to the lrelease call. Not set by default.
embed_translations	Embed the generated translations from `lrelease` in the executable, under QM_FILES_RESOURCE_PREFIX. Requires `lrelease` to be set, too. Not set by default.
create_libtool	Create a libtool .la file for the currently built library.
create_pc	Create a pkg-config .pc file for the currently built library.
no_batch	NMake only: Turn off generation of NMake batch rules or inference rules.
skip_target_version_ext	Suppress the automatic version number appended to the DLL file name on Windows.
suppress_vcproj_warnings	Suppress warnings of the VS project generator.
windeployqt	Automatically invoke windeployqt after linking, and add the output as deployment items.
dont_recurse	Suppress qmake recursion for the current subproject.
no_include_pwd	Do not add the current directory to INCLUDEPATHS.
compile_included_sources	By default, qmake does not compile source files that are included in other source files. This option disables this behavior.

When you use the `debug_and_release` option (which is the default under Windows), the project will be processed three times: one time to produce a "meta" Makefile, and two more times to produce a Makefile.Debug

This makes it possible to perform build-specific tasks. For example:

```
build_pass:CONFIG(debug, debug|release) {
    unix: TARGET = $$join(TARGET,,,_debug)
    else: TARGET = $$join(TARGET,,,d)
}
```

As an alternative to manually writing build type conditionals, some variables offer build-specific variants, for example QMAKE_LFLAGS_RELEASE in addition to the general QMAKE_LFLAGS. These should be used when available.

The meta Makefile makes the sub-builds invokable via the debug and `release` targets, and a combined build via the `all` target. When the `build_all` CONFIG option is used, the combined build is the default. Otherwise, the last specified CONFIG option from the set (debug, `release`) determines the default. In this case, you can explicitly invoke the `all` target to build both configurations at once:

```
make all
```

Note: The details are slightly different when producing Visual Studio and Xcode projects.

When linking a library, qmake relies on the underlying platform to know what other libraries this library links against. However, if linking statically, qmake will not get this information unless we use the following CONFIG options:

Option	Description
create_prl	This option enables qmake to track these dependencies. When this option is enabled, qmake will create a file with the extension `.prl` which will save meta-information about the library (see Library Dependencies for more info).
link_prl	When this option is enabled, qmake will process all libraries linked to by the application and find their meta-information (see Library Dependencies for more info).
no_install_prl	This option disables the generation of installation rules for generated .prl files.

Note: The `create_prl` option is required when *building* a static library, while `link_prl` is required when *using* a static library.

The following options define the application or library type:

Option	Description
qt	The target is a Qt application or library and requires the Qt library and header files. The proper include and library paths for the Qt library will automatically be added to the project. This is defined by default, and can be fine-tuned with the `\l{#qt}{QT}` variable.
x11	The target is an X11 application or library. The proper include paths and libraries will automatically be added to the project.

Option	Description

insignificant_test	The exit code of the automated test will be ignored. Only relevant if `testcase` is also set.
windows	The target is a Win32 window application (app only). The proper include paths, compiler flags and libraries will automatically be added to the project.
console	The target is a Win32 console application (app only). The proper include paths, compiler flags and libraries will automatically be added to the project. Consider using the option `cmdline` for cross-platform applications.
cmdline	The target is a cross-platform command line application. On Windows, this implies `CONFIG += console`. On macOS, this implies `CONFIG -= app_bundle`.
shared	The target is a shared object/DLL. The proper include paths, compiler flags and libraries will automatically be added to the project. Note that `dll` can also be used on all platforms; a shared library file with the appropriate suffix for the target platform (.dll or .so) will be created.
dll	
static	The target is a static library (lib only). The proper compiler flags will automatically be added to the project.
staticlib	
plugin	The target is a plugin (lib only). This enables dll as well.
designer	The target is a plugin for *Qt Designer*.
no_lflags_merge	Ensures that the list of libraries stored in the `LIBS` variable is not reduced to a list of unique values before it is used.
metatypes	Create a `<name>_metatypes.json` file for the current project. `<name>` is the all lowercase base name of TARGET.
qmltypes	Automatically register QML types defined in C++. For more information, see Defining QML Types from C++. Also, create a `<template>.qmltypes` file for the current project. `<template>` will be `plugins` (plural, for historical reasons) if `plugin` is set, or the value of TEMPLATE otherwise. `qmltypes` implies `metatypes`.

These options define specific features on Windows only:

Option	Description
flat	When using the vcapp template this will put all the source files into the source group and the header files into the header group regardless of what directory they reside in. Turning this option off will group the files within the source/header group depending on the directory they reside. This is turned on by default.
embed_manifest_dll	Embeds a manifest file in the DLL created as part of a library project.
embed_manifest_exe	Embeds a manifest file in the EXE created as part of an application project.

See Platform Notes for more information about the options for embedding manifest files.

The following options take an effect only on macOS:

Option	Description
app_bundle	Puts the executable into a bundle (this is the default).
lib_bundle	Puts the library into a library bundle.
plugin_bundle	Puts the plugin into a plugin bundle. This value is not supported by the Xcode project generator.

The following options take an effect only on Linux/Unix platforms:

Option	Description
largefile	Includes support for large files.
separate_debug_info	Puts debugging information for libraries in separate files.

The CONFIG variable will also be checked when resolving scopes. You may assign anything to this variable.

For example:

```
CONFIG += console newstuff
...
newstuff {
    SOURCES += new.cpp
    HEADERS += new.h
}
```

DEFINES

qmake adds the values of this variable as compiler C preprocessor macros (-D option).

For example:

```
DEFINES += USE_MY_STUFF
```

DEFINES_DEBUG

Specifies preprocessor defines for the debug configuration. The values of this variable get added to DEFINES after the project is loaded. This variable is typically set in qmake.conf and rarely needs to be modified.

This variable was introduced in Qt 5.13.2.

DEFINES_RELEASE

Specifies preprocessor defines for the release configuration. The values of this variable get added to DEFINES after the project is loaded. This variable is typically set in qmake.conf and rarely needs to be modified.

> **Note:** For MSVC mkspecs, this variable contains the value NDEBUG by default.

This variable was introduced in Qt 5.13.2.

DEF_FILE

Specifies a `.def` file to be included in the project.

DEPENDPATH

Specifies a list of directories for qmake to scan, to resolve dependencies. This variable is used when qmake crawls through the header files that you `#include` in your source code.

DESTDIR

Specifies where to put the target file.

For example:

```
DESTDIR = ../../lib
```

> **Note:** The list of supported characters can depend on the used build tool. In particular, parentheses do not work with `make`.

DISTFILES

Specifies a list of files to be included in the dist target. This feature is supported by UnixMake specs only.

For example:

```
DISTFILES += ../program.txt
```

DLLDESTDIR

> **Note:** This variable applies only to Windows targets.

Specifies where to copy the target dll.

EXTRA_TRANSLATIONS

Specifies a list of translation (.ts) files that contain translations of the user interface text into non-native languages.

In contrast to TRANSLATIONS, translation files in `EXTRA_TRANSLATIONS` will be processed only by lrelease, not lupdate.

See the Qt Linguist Manual for more information about internationalization (i18n) and localization (l10n) with Qt.

FORMS

Specifies the UI files (see Qt Designer Manual) to be processed by `uic` before compiling. All dependencies, headers and source files required to build these UI files will automatically be added to the project.

For example:

```
FORMS = mydialog.ui \
    mywidget.ui \
        myconfig.ui
```

GUID

Specifies the GUID that is set inside a `.vcproj` file. The GUID is usually randomly determined. However, should you require a fixed GUID, it can be set using this variable.

This variable is specific to `.vcproj` files only; it is ignored otherwise.

HEADERS

Defines the header files for the project.

qmake automatically detects whether moc is required by the classes in the headers, and adds the appropriate dependencies and files to the project for generating and linking the moc files.

For example:

```
HEADERS = myclass.h \
          login.h \
          mainwindow.h
```

See also SOURCES.

ICON

This variable is used only on Mac OS to set the application icon. Please see the application icon documentation for more information.

IDLSOURCES

This variable is used only on Windows for the Visual Studio project generation to put the specified files in the Generated Files folder.

INCLUDEPATH

Specifies the #include directories which should be searched when compiling the project.

For example:

```
INCLUDEPATH = c:/msdev/include d:/stl/include
```

To specify a path containing spaces, quote the path using the technique described in Whitespace.

```
win32:INCLUDEPATH += "C:/mylibs/extra headers"
unix:INCLUDEPATH += "/home/user/extra headers"
```

INSTALLS

Specifies a list of resources that will be installed when `make install` or a similar installation procedure is executed. Each item in the list is typically defined with attributes that provide information about where it will be installed.

For example, the following `target.path` definition describes where the build target will be installed, and the `INSTALLS` assignment adds the build target to the list of existing resources to be installed:

```
target.path += $$[QT_INSTALL_PLUGINS]/imageformats
INSTALLS += target
```

`INSTALLS` has a `.CONFIG` member that can take several values:

Value	Description
no_check_exist	If not set, qmake looks to see if the files to install actually exist. If these files don't exist, qmake doesn't create the install rule. Use this config value if you need to install files that are generated as part of your build process, like HTML files created by qdoc.
nostrip	If set, the typical Unix strip functionality is turned off and the debug information will remain in the binary.
executable	On Unix, this sets the executable flag.
no_build	When you do a `make install`, and you don't have a build of the project yet, the project is first built, and then installed. If you don't want this behavior, set this config value to ensure that the build target is not added as a dependency to the install target.
no_default_install	A project has a top-level project target where, when you do a `make install`, everything is installed. But, if you have an install target with this config value set, it's not installed by default. You then have to explicitly say `make install_<file>`.

For more information, see Installing Files.

JAVA_HOME

> **Note:** This variable is useful only to Android targets.

Specifies the JDK/OpenJDK installation path used for building the project.

LEXIMPLS

Specifies a list of Lex implementation files. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

LEXOBJECTS

Specifies the names of intermediate Lex object files. The value of this variable is typically handled by qmake and rarely needs to be modified.

LEXSOURCES

Specifies a list of Lex source files. All dependencies, headers and source files will automatically be added to the project for building these lex files.

For example:

```
LEXSOURCES = lexer.l
```

LIBS

Specifies a list of libraries to be linked into the project. If you use the Unix `-l` (library) and -L (library path) flags, qmake handles the libraries correctly on Windows (that is, passes the full path of the library to the linker). The library must exist for qmake to find the directory where a `-l` lib is located.

For example:

```
unix:LIBS += -L/usr/local/lib -lmath
win32:LIBS += c:/mylibs/math.lib
```

To specify a path containing spaces, quote the path using the technique described in Whitespace.

```
win32:LIBS += "C:/mylibs/extra libs/extra.lib"
unix:LIBS += "-L/home/user/extra libs" -lextra
```

behavior, add the `no_lflags_merge` option to the CONFIG variable:

```
CONFIG += no_lflags_merge
```

LIBS_PRIVATE

Specifies a list of libraries to be linked privately into the project. The behavior of this variable is identical to LIBS, except that shared library projects built for Unix do not expose these dependencies in their link interface.

The effect of this is that if project C depends on library B which depends on library A privately, but C also wants to use symbols from A directly, it needs to link to A explicitly. Put differently, libraries linked privately are not exposed transitively at build time.

LITERAL_HASH

This variable is used whenever a literal hash character (#) is needed in a variable declaration, perhaps as part of a file name or in a string passed to some external application.

For example:

```
# To include a literal hash character, use the $$LITERAL_HASH variable:
urlPieces = http://doc.qt.io/qt-5/qtextdocument.html pageCount
message($$join(urlPieces, $$LITERAL_HASH))
```

By using `LITERAL_HASH` in this way, the # character can be used to construct a URL for the `message()` function to print to the console.

MAKEFILE

Specifies the name of the generated Makefile. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

MAKEFILE_GENERATOR

Specifies the name of the Makefile generator to use when generating a Makefile. The value of this variable is typically handled internally by qmake and rarely needs to be modified.

MSVCPROJ_*

These variables are handled internally by qmake and should not be modified or utilized.

MOC_DIR

For example:

```
unix:MOC_DIR = ../myproject/tmp
win32:MOC_DIR = c:/myproject/tmp
```

OBJECTIVE_HEADERS

Defines the Objective-C++ header files for the project.

qmake automatically detects whether moc is required by the classes in the headers, and adds the appropriate dependencies and files to the project for generating and linking the moc files.

This is similar to the HEADERS variable, but will let the generated moc files be compiled with the Objective-C++ compiler.

See also OBJECTIVE_SOURCES.

OBJECTIVE_SOURCES

Specifies the names of all Objective-C/C++ source files in the project.

This variable is now obsolete, Objective-C/C++ files (.m and .mm) can be added to the SOURCES variable.

See also OBJECTIVE_HEADERS.

OBJECTS

This variable is automatically populated from the SOURCES variable. The extension of each source file is replaced by .o (Unix) or .obj (Win32). You can add objects to the list.

OBJECTS_DIR

Specifies the directory where all intermediate objects should be placed.

For example:

```
unix:OBJECTS_DIR = ../myproject/tmp
win32:OBJECTS_DIR = c:/myproject/tmp
```

POST_TARGETDEPS

Lists the libraries that the target depends on. Some backends, such as the generators for Visual Studio and Xcode project files, do not support this variable. Generally, this variable is supported internally by these build tools, and it is useful for explicitly listing dependent static libraries.

This list is placed after all builtin (and $$PRE_TARGETDEPS) dependencies

PRE_TARGETDEPS

Lists libraries that the target depends on. Some backends, such as the generators for Visual Studio and Xcode project files, do not support this variable. Generally, this variable is supported internally by these build tools, and it is useful for explicitly listing dependent static libraries.

This list is placed before all builtin dependencies.

PRECOMPILED_HEADER

Indicates the header file for creating a precompiled header file, to increase the compilation speed of a project. Precompiled headers are currently only supported on some platforms (Windows - all MSVC project types, Apple - Xcode, Makefile, Unix - gcc 3.3 and up).

PWD

Specifies the full path leading to the directory containing the current file being parsed. This can be useful to refer to files within the source tree when writing project files to support shadow builds.

See also _PRO_FILE_PWD_.

> **Note:** Do not attempt to overwrite the value of this variable.

OUT_PWD

Specifies the full path leading to the directory where qmake places the generated Makefile.

> **Note:** Do not attempt to overwrite the value of this variable.

QM_FILES_RESOURCE_PREFIX

Specifies the directory in the resource system where `.qm` files will be made available by CONFIG += embed_translations.

The default is `:/i18n/`.

QM_FILES_INSTALL_PATH

Specifies the target directory `.qm` files generated by CONFIG += lrelease will be installed to. Does not have any effect if CONFIG += embed_translations is set.

QML_IMPORT_PATH

This variable is only used by Qt Creator. If you have an extra module that is kept outside of your Qt installation, you can specify its path here.

QMLPATHS

Expects a list of import paths that point to root directories of trees of QML modules. For example, if you have a custom location for your QML modules, you can specify it here.

Note: The path entries for QMLPATHS point to root directories of trees of QML modules. This is the concept of import paths the QML engine understands. You can pass the same paths via the `QML_IMPORT_PATH` *environment* variable to your QML application, but they are *different* from the expected contents of the QML_IMPORT_PATH *qmake* variable. The latter expects paths to *individual* modules to be processed by Qt Creator only.

Note: The contents of QMLPATHS are *not* automatically passed to your application. Rather, they are only used at build time. In particular, qmlimportscanner uses them to find any QML modules it may need to mark as imported by your application.

QMAKE

Specifies the name of the qmake program itself and is placed in generated Makefiles. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKESPEC

A system variable that contains the full path of the qmake configuration that is used when generating Makefiles. The value of this variable is automatically computed.

Note: Do not attempt to overwrite the value of this variable.

QMAKE_AR_CMD

Note: This variable is used on Unix platforms only.

Specifies the command to execute when creating a shared library. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_BUNDLE_DATA

Note: This variable is used on macOS, iOS, tvOS, and watchOS only.

Specifies the data that will be installed with a library bundle, and is often used to specify a collection of header files.

For example, the following lines add `path/to/header_one.h` and `path/to/header_two.h` to a group

```
FRAMEWORK_HEADERS.version = Versions
FRAMEWORK_HEADERS.files = path/to/header_one.h path/to/header_two.h
FRAMEWORK_HEADERS.path = Headers
QMAKE_BUNDLE_DATA += FRAMEWORK_HEADERS
```

The last line adds the information about the headers to the collection of resources that will be installed with the library bundle.

Library bundles are created when the `lib_bundle` option is added to the CONFIG variable.

See Platform Notes for more information about creating library bundles.

A project can also use this variable to bundle application translation files. The exact syntax depends on whether the project is using Xcode's legacy build system or its new build system.

For example when the following project snippet is built using the legacy build system:

```
translations_en.files = $$PWD/en.lproj/InfoPlist.strings
translations_en.path = en.lproj
QMAKE_BUNDLE_DATA += translations_en
```

Xcode will ignore the original location of `InfoPlist.strings` and the file will placed into the bundle `Resources` directory under the provided `translations_en.path` path, so `Resources/en.lproj/InfoPlist.strings`

With the new build system, the relative location of the file is preserved, which means the file will incorrectly be placed under `Resources/en.lproj/en.lproj/InfoPlist.strings`

To ensure correct file placement, the project can either move the original file not to be in a sub-directory or it can choose not to specify the `translations_en.path` variable.

```
# Approach 1
translations_en.files = $$PWD/InfoPlist.strings
translations_en.path = en.lproj

# Approach 2
translations_de.files = $$PWD/de.lproj/InfoPlist.strings

QMAKE_BUNDLE_DATA += translations_en translations_de
```

See QTBUG-98417 for more details on how the Xcode build system changed its behavior in bundling translation files.

QMAKE_BUNDLE_EXTENSION

> **Note:** This variable is used on macOS, iOS, tvOS, and watchOS only.

For example, the following definition will result in a framework with the `.myframework` extension:

```
QMAKE_BUNDLE_EXTENSION = .myframework
```

QMAKE_CC

Specifies the C compiler that will be used when building projects containing C source code. Only the file name of the compiler executable needs to be specified as long as it is on a path contained in the PATH variable when the Makefile is processed.

QMAKE_CFLAGS

Specifies the C compiler flags for building a project. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified. The flags specific to debug and release modes can be adjusted by modifying the QMAKE_CFLAGS_DEBUG and QMAKE_CFLAGS_RELEASE variables, respectively.

QMAKE_CFLAGS_DEBUG

Specifies the C compiler flags for debug builds. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_RELEASE

Specifies the C compiler flags for release builds. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_RELEASE_WITH_DEBUGINFO

Specifies the C compiler flags for release builds where `force_debug_info` is set in CONFIG. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_SHLIB

> **Note:** This variable is used on Unix platforms only.

Specifies the compiler flags for creating a shared library. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_THREAD

Specifies the compiler flags for creating a multi-threaded application. The value of this variable is typically handled

QMAKE_CFLAGS_WARN_OFF

This variable is used only when the `warn_off` CONFIG option is set. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_WARN_ON

This variable is used only when the `warn_on` CONFIG option is set. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CLEAN

Specifies a list of generated files (by moc and uic, for example) and object files to be removed by `make clean`.

QMAKE_CXX

Specifies the C++ compiler that will be used when building projects containing C++ source code. Only the file name of the compiler executable needs to be specified as long as it is on a path contained in the PATH variable when the Makefile is processed.

QMAKE_CXXFLAGS

Specifies the C++ compiler flags for building a project. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified. The flags specific to debug and release modes can be adjusted by modifying the QMAKE_CXXFLAGS_DEBUG and QMAKE_CXXFLAGS_RELEASE variables, respectively.

QMAKE_CXXFLAGS_DEBUG

Specifies the C++ compiler flags for debug builds. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_RELEASE

Specifies the C++ compiler flags for release builds. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_RELEASE_WITH_DEBUGINFO

Specifies the C++ compiler flags for release builds where `force_debug_info` is set in CONFIG. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_SHLIB

Specifies the C++ compiler flags for creating a shared library. The value of this variable is typically handled by qmake

QMAKE_CXXFLAGS_THREAD

Specifies the C++ compiler flags for creating a multi-threaded application. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_WARN_OFF

Specifies the C++ compiler flags for suppressing compiler warnings. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_WARN_ON

Specifies C++ compiler flags for generating compiler warnings. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_DEVELOPMENT_TEAM

> **Note:** This variable is used on macOS, iOS, tvOS, and watchOS only.

The identifier of a development team to use for signing certificates and provisioning profiles.

QMAKE_DISTCLEAN

Specifies a list of files to be removed by `make distclean`.

QMAKE_EXTENSION_SHLIB

Contains the extension for shared libraries. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

> **Note:** Platform-specific variables that change the extension override the contents of this variable.

QMAKE_EXTENSION_STATICLIB

Contains the extension for shared static libraries. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_EXT_MOC

Contains the extension used on included moc files.

See also File Extensions.

QMAKE_EXT_UI

Contains the extension used on *Qt Designer* UI files.

See also File Extensions.

QMAKE_EXT_PRL

Contains the extension used on created PRL files.

See also File Extensions, Library Dependencies.

QMAKE_EXT_LEX

Contains the extension used on files given to Lex.

See also File Extensions, LEXSOURCES.

QMAKE_EXT_YACC

Contains the extension used on files given to Yacc.

See also File Extensions, YACCSOURCES.

QMAKE_EXT_OBJ

Contains the extension used on generated object files.

See also File Extensions.

QMAKE_EXT_CPP

Contains suffixes for files that should be interpreted as C++ source code.

See also File Extensions.

QMAKE_EXT_H

Contains suffixes for files which should be interpreted as C header files.

See also File Extensions.

QMAKE_EXTRA_COMPILERS

Specifies a list of additional compilers or preprocessors.

See also Adding Compilers.

Specifies a list of additional qmake targets.

See also Adding Custom Targets.

QMAKE_FAILED_REQUIREMENTS

Contains the list of failed requirements. The value of this variable is set by qmake and cannot be modified.

See also requires() and REQUIRES.

QMAKE_FRAMEWORK_BUNDLE_NAME

> **Note:** This variable is used on macOS, iOS, tvOS, and watchOS only.

In a framework project, this variable contains the name to be used for the framework that is built.

By default, this variable contains the same value as the TARGET variable.

See Creating Frameworks for more information about creating frameworks and library bundles.

QMAKE_FRAMEWORK_VERSION

> **Note:** This variable is used on macOS, iOS, tvOS, and watchOS only.

For projects where the build target is a macOS, iOS, tvOS, or watchOS framework, this variable is used to specify the version number that will be applied to the framework that is built.

By default, this variable contains the same value as the VERSION variable.

See Creating Frameworks for more information about creating frameworks.

QMAKE_HOST

Provides information about the host machine running qmake. For example, you can retrieve the host machine architecture from `QMAKE_HOST.arch`.

Keys	Values
.arch	Host architecture
.os	Host OS
.cpu_count	Number of available cpus
.name	Host computer name
.version	Host OS version number
.version_string	Host OS version string

```
    message("Host is 64bit")
    ...
}
```

QMAKE_INCDIR

Specifies the list of system header paths that are appended to INCLUDEPATH. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR_EGL

Specifies the location of EGL header files to be added to INCLUDEPATH when building a target with OpenGL/ES or OpenVG support. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR_OPENGL

Specifies the location of OpenGL header files to be added to INCLUDEPATH when building a target with OpenGL support. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

If the OpenGL implementation uses EGL (most OpenGL/ES systems), then QMAKE_INCDIR_EGL may also need to be set.

QMAKE_INCDIR_OPENGL_ES2

This variable specifies the location of OpenGL header files to be added to INCLUDEPATH when building a target with OpenGL ES 2 support.

The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

If the OpenGL implementation uses EGL (most OpenGL/ES systems), then QMAKE_INCDIR_EGL may also need to be set.

QMAKE_INCDIR_OPENVG

Specifies the location of OpenVG header files to be added to INCLUDEPATH when building a target with OpenVG support. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

If the OpenVG implementation uses EGL then QMAKE_INCDIR_EGL may also need to be set.

QMAKE_INCDIR_X11

> **Note:** This variable is used on Unix platforms only.

Specifies the location of X11 header file paths to be added to INCLUDEPATH when building an X11 target. The value

Qt DOCUMENTATION

QMAKE_INFO_PLIST

Note: This variable is used on macOS, iOS, tvOS, and watchOS platforms only.

Specifies the name of the property list file, `.plist`, you would like to include in your macOS, iOS, tvOS, and watchOS application bundle.

In the `.plist` file, you can define some variables which qmake will replace with the relevant values:

Placeholder(s)	Effect
`${PRODUCT_BUNDLE_IDENTIFIER}`, `@BUNDLEIDENTIFIER@`	Expands to the target bundle's bundle identifier string, for example: `com.example.myapp`. Determined by concatenating the values of QMAKE_TARGET_BUNDLE_PREFIX and QMAKE_BUNDLE, separated by a full stop (`.`).
`${EXECUTABLE_NAME}`, `@EXECUTABLE@`, `@LIBRARY@`	Equivalent to the value of QMAKE_APPLICATION_BUNDLE_NAME, QMAKE_PLUGIN_BUNDLE_NAME, or QMAKE_FRAMEWORK_BUNDLE_NAME (depending on the type of target being created), or TARGET if none of the previous values are set.
`${ASSETCATALOG_COMPILER_APPICON_NAME}`, `@ICON@`	Expands to the value of ICON.
`${QMAKE_PKGINFO_TYPEINFO}`, `@TYPEINFO@`	Expands to the value of QMAKE_PKGINFO_TYPEINFO.
`${QMAKE_FULL_VERSION}`, `@FULL_VERSION@`	Expands to the value of VERSION expressed with three version components.
`${QMAKE_SHORT_VERSION}`, `@SHORT_VERSION@`	Expands to the value of VERSION expressed with two version components.
`${MACOSX_DEPLOYMENT_TARGET}`	Expands to the value of QMAKE_MACOSX_DEPLOYMENT_TARGET.
`${IPHONEOS_DEPLOYMENT_TARGET}`	Expands to the value of QMAKE_IPHONEOS_DEPLOYMENT_TARGET.
`${TVOS_DEPLOYMENT_TARGET}`	Expands to the value of QMAKE_TVOS_DEPLOYMENT_TARGET.
`${WATCHOS_DEPLOYMENT_TARGET}`	Expands to the value of QMAKE_WATCHOS_DEPLOYMENT_TARGET.
`${IOS_LAUNCH_SCREEN}`	Expands to the value of QMAKE_IOS_LAUNCH_SCREEN.

Note: When using the Xcode generator, the above `${var}`-style placeholders are replaced directly by the Xcode build system and are not handled by qmake. The `@var@` style placeholders work only with the qmake Makefile generators and not with the Xcode generator.

If building for iOS, and the `.plist` file contains the key `NSPhotoLibraryUsageDescription`, qmake will include an additional plugin to the build that adds photo access support (to, e.g., QFile/QFileDialog). See Info.plist

Note: Most of the time, the default `Info.plist` is good enough.

QMAKE_IOS_DEPLOYMENT_TARGET

Note: This variable is used on the iOS platform only.

Specifies the hard minimum version of iOS that the application supports.

For more information, see Expressing Supported iOS Versions.

QMAKE_IOS_LAUNCH_SCREEN

Note: This variable is used on the iOS platform only.

Specifies the launch screen that is used by the application. If this is not set then a default launch screen is used.

QMAKE_LFLAGS

Specifies a general set of flags that are passed to the linker. If you need to change the flags used for a particular platform or type of project, use one of the specialized variables for that purpose instead of this variable.

QMAKE_LFLAGS_CONSOLE

Note: This variable is used on Windows only.

Specifies the linker flags for building console programs. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_DEBUG

Specifies the linker flags for debug builds. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_PLUGIN

Specifies the linker flags for building plugins. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_RPATH

Specifies the linker flags needed to use the values from QMAKE_RPATHDIR.

The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_REL_RPATH

Specifies the linker flags needed to enable relative paths in QMAKE_RPATHDIR.

The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_REL_RPATH_BASE

Specifies the string the dynamic linker understands to be the location of the referring executable or library.

The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_RPATHLINK

Specifies the linker flags needed to use the values from QMAKE_RPATHLINKDIR.

The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_RELEASE

Specifies the linker flags for release builds. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_RELEASE_WITH_DEBUGINFO

Specifies the linker flags for release builds where `force_debug_info` is set in `CONFIG`. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_APP

Specifies the linker flags for building applications. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_SHLIB

Specifies the linker flags used for building shared libraries. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_SONAME

Specifies the linker flags for setting the name of shared objects, such as .so or .dll. The value of this variable is

QMAKE_LFLAGS_THREAD

Specifies the linker flags for building multi-threaded projects. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_WINDOWS

> **Note:** This variable is used on Windows only.

Specifies the linker flags for building Windows GUI projects (that is, non-console applications). The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR

Specifies a list of library search paths for all projects. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

To specify additional search paths in project files, use LIBS like that, instead:

```
LIBS += -L/path/to/libraries
```

QMAKE_LIBDIR_POST

Specifies a list of system library search paths for all projects. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR_FLAGS

> **Note:** This variable is used on Unix platforms only.

Specifies the location of all library directories with -L prefixed. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR_EGL

Specifies the location of the EGL library directory, when EGL is used with OpenGL/ES or OpenVG. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR_OPENGL

Specifies the location of the OpenGL library directory. The value of this variable is typically handled by qmake or

be set.

QMAKE_LIBDIR_OPENVG

Specifies the location of the OpenVG library directory. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

If the OpenVG implementation uses EGL, then QMAKE_LIBDIR_EGL may also need to be set.

QMAKE_LIBDIR_X11

> **Note:** This variable is used on Unix platforms only.

Specifies the location of the X11 library directory. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIBS

Specifies additional libraries each project needs to link against. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

To specify libraries in a project file, use LIBS instead.

QMAKE_LIBS_PRIVATE

Specifies additional private libraries each project needs to link against. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

To specify private libraries in a library project file, use LIBS_PRIVATE instead.

QMAKE_LIBS_EGL

Specifies all EGL libraries when building Qt with OpenGL/ES or OpenVG. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified. The usual value is `-lEGL`.

QMAKE_LIBS_OPENGL

Specifies all OpenGL libraries. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

If the OpenGL implementation uses EGL (most OpenGL/ES systems), then QMAKE_LIBS_EGL may also need to be set.

QMAKE_LIBS_OPENGL_ES1, QMAKE_LIBS_OPENGL_ES2

If the OpenGL implementation uses EGL (most OpenGL/ES systems), then QMAKE_LIBS_EGL may also need to be set.

QMAKE_LIBS_OPENVG

Specifies all OpenVG libraries. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified. The usual value is `-lOpenVG`.

Some OpenVG engines are implemented on top of OpenGL. This will be detected at configure time and QMAKE_LIBS_OPENGL will be implicitly added to QMAKE_LIBS_OPENVG wherever the OpenVG libraries are linked.

If the OpenVG implementation uses EGL, then QMAKE_LIBS_EGL may also need to be set.

QMAKE_LIBS_THREAD

> **Note:** This variable is used on Unix platforms only.

Specifies all libraries that need to be linked against when building a multi-threaded target. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_X11

> **Note:** This variable is used on Unix platforms only.

Specifies all X11 libraries. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LIB_FLAG

This variable is not empty if the `lib` template is specified. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LINK

Specifies the linker that will be used when building application based projects. Only the file name of the linker executable needs to be specified as long as it is on a path contained in the PATH variable when the Makefile is processed. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LINK_SHLIB_CMD

Specifies the command to execute when creating a shared library. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

Specifies the command to execute when creating a link to a shared library. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_LRELEASE_FLAGS

List of additional options passed to lrelease when enabled through CONFIG += lrelease.

QMAKE_OBJECTIVE_CFLAGS

Specifies the Objective C/C++ compiler flags for building a project. These flags are used in addition to QMAKE_CFLAGS and QMAKE_CXXFLAGS.

QMAKE_POST_LINK

Specifies the command to execute after linking the TARGET together. This variable is normally empty and therefore nothing is executed.

> **Note:** This variable takes no effect on Xcode projects.

QMAKE_PRE_LINK

Specifies the command to execute before linking the TARGET together. This variable is normally empty and therefore nothing is executed.

> **Note:** This variable takes no effect on Xcode projects.

QMAKE_PROJECT_NAME

> **Note:** This variable is used for Visual Studio project files only.

Determines the name of the project when generating project files for IDEs. The default value is the target name. The value of this variable is typically handled by qmake and rarely needs to be modified.

QMAKE_PROVISIONING_PROFILE

> **Note:** This variable is used on macOS, iOS, tvOS, and watchOS only.

The UUID of a valid provisioning profile. Use in conjunction with QMAKE_DEVELOPMENT_TEAM to specify the provisioning profile.

QMAKE_MAC_SDK

This variable is used on macOS when building universal binaries.

QMAKE_MACOSX_DEPLOYMENT_TARGET

> **Note:** This variable is used on the macOS platform only.

Specifies the hard minimum version of macOS that the application supports.

For more information, see macOS Supported Versions.

QMAKE_MAKEFILE

Specifies the name of the Makefile to create. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_QMAKE

Contains the absolute path of the qmake executable.

> **Note:** Do not attempt to overwrite the value of this variable.

QMAKE_RESOURCE_FLAGS

This variable is used to customize the list of options passed to the Resource Compiler in each of the build rules where it is used. For example, the following line ensures that the `-threshold` and `-compress` options are used with particular values each time that `rcc` is invoked:

```
QMAKE_RESOURCE_FLAGS += -threshold 0 -compress 9
```

QMAKE_RPATHDIR

> **Note:** This variable is used on Unix platforms only.

Specifies a list of library paths that are added to the executable at link time so that the paths will be preferentially searched at runtime.

When relative paths are specified, qmake will mangle them into a form understood by the dynamic linker to be relative to the location of the referring executable or library. This is supported only by some platforms (currently

QMAKE_RPATHLINKDIR

Specifies a list of library paths for the static linker to search for implicit dependencies of shared libraries. For more information, see the manual page for `ld(1)`.

QMAKE_RUN_CC

Specifies the individual rule needed to build an object. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_RUN_CC_IMP

Specifies the individual rule needed to build an object. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_RUN_CXX

Specifies the individual rule needed to build an object. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_RUN_CXX_IMP

Specifies the individual rule needed to build an object. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_SONAME_PREFIX

If defined, the value of this variable is used as a path to be prepended to the built shared library's SONAME identifier. The SONAME is the identifier that the dynamic linker will later use to reference the library. In general, this reference may be a library name or full library path. On macOS, iOS, tvOS, and watchOS, the path may be specified relatively using the following placeholders:

Placeholder	Effect
@rpath	Expands to paths defined by LC_RPATH mach-o commands in the current process executable or the referring libraries.
@executable_path	Expands to the current process executable location.
@loader_path	Expands to the referring executable or library location.

In most cases, using `@rpath` is sufficient and recommended:

```
# <project root>/project.pro
QMAKE_SONAME_PREFIX = @rpath
```

```
# <project root>/project.pro
QMAKE_SONAME_PREFIX = @executable_path/../Frameworks
QMAKE_SONAME_PREFIX = @loader_path/Frameworks
QMAKE_SONAME_PREFIX = /Library/Frameworks
```

For more information, see dyld documentation on dynamic library install names.

QMAKE_TARGET

Specifies the name of the project target. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

QMAKE_TARGET_COMPANY

Windows only. Specifies the company for the project target; this is used where applicable for putting the company name in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_TARGET_DESCRIPTION

Windows only. Specifies the description for the project target; this is used where applicable for putting the description in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_TARGET_COPYRIGHT

Windows only. Specifies the copyright information for the project target; this is used where applicable for putting the copyright information in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_TARGET_PRODUCT

Windows only. Specifies the product for the project target; this is used where applicable for putting the product in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_TARGET_ORIGINAL_FILENAME

Windows only. Specifies the original file name for the project target; this is used where applicable for putting the original file name in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

Windows only. Specifies the internal name for the project target; this is used where applicable for putting the internal name in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_TARGET_COMMENTS

Windows only. Specifies the comments for the project target; this is used where applicable for putting the comments in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_TARGET_TRADEMARKS

Windows only. Specifies the trademark information for the project target; this is used where applicable for putting the trademark information in the application's properties. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

QMAKE_MANIFEST

Windows only. Specifies the manifest file for the project target. This is only utilized if the RC_FILE and RES_FILE variables are not set. Don't forget to remove embed_manifest_exe and embed_manifest_dll from the CONFIG variable, otherwise it will conflict with the compiler generated one.

QMAKE_TVOS_DEPLOYMENT_TARGET

> **Note:** This variable is used on the tvOS platform only.

Specifies the hard minimum version of tvOS that the application supports.

For more information, see Expressing Supported iOS Versions.

QMAKE_UIC_FLAGS

This variable is used to customize the list of options passed to the User Interface Compiler in each of the build rules where it is used.

QMAKE_WATCHOS_DEPLOYMENT_TARGET

> **Note:** This variable is used on the watchOS platform only.

Specifies the hard minimum version of watchOS that the application supports.

For more information, see Expressing Supported iOS Versions.

QML_IMPORT_MAJOR_VERSION

see Defining QML Types from C++.

QML_IMPORT_MINOR_VERSION

When automatically registering QML types defined in C++, register an additional version of the module using this minor version. Generally, minor versions to be registered are inferred from the meta objects.

You can use this variable if the meta objects have not changed and you still want to import a QML module with a newer minor version number. For example, `MyModule` metaobjects are at `1.1` level, but you want to import the module as `1.3`.

QML_IMPORT_VERSION

Specifies QML_IMPORT_MAJOR_VERSION and QML_IMPORT_MINOR_VERSION as a `<major>.<minor>` version string.

QML_IMPORT_NAME

Specifies the module name to be used for automatically generated QML type registrations. For more information, see Defining QML Types from C++.

QML_FOREIGN_METATYPES

Specifies further JSON files with metatypes to be considered when generating qmltypes files. Use this when external libraries provide types that are exposed to QML, either directly or as base types or properties of other types. Qt types will automatically be considered and don't have to be added here.

QT

Specifies the Qt modules that are used by your project. For the value to add for each module, see the module documentation.

At the C++ implementation level, using a Qt module makes its headers available for inclusion and causes it to be linked to the binary.

By default, `QT` contains `core` and `gui`, ensuring that standard GUI applications can be built without further configuration.

If you want to build a project *without* the Qt GUI module, you need to exclude the `gui` value with the "-=" operator. The following line will result in a minimal Qt project being built:

```
QT -= gui # Only the core module is used.
```

If your project is a *Qt Designer* plugin, use the value `uiplugin` to specify that the project is to be built as a library, but with specific plugin support for *Qt Designer*. For more information, see Building and Installing the Plugin.

Specifies a list of names of static Qt plugins that are to be linked with an application so that they are available as built-in resources.

qmake automatically adds the plugins that are typically needed by the used Qt modules (see QT). The defaults are tuned towards an optimal out-of-the-box experience. See Static Plugins for a list of available plugins, and ways to override the automatic linking.

This variable currently has no effect when linking against a shared/dynamic build of Qt, or when linking libraries. It may be used for deployment of dynamic plugins at a later time.

QT_VERSION

Contains the current version of Qt.

QT_MAJOR_VERSION

Contains the current major version of Qt.

QT_MINOR_VERSION

Contains the current minor version of Qt.

QT_PATCH_VERSION

Contains the current patch version of Qt.

RC_FILE

Windows only. Specifies the name of the Windows resource file (.rc) for the target. See Adding Windows Resource Files.

RC_CODEPAGE

Windows only. Specifies the codepage that should be specified in a generated .rc file. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

RC_DEFINES

Windows only. qmake adds the values of this variable as RC preprocessor macros (/d option). If this variable is not set, the DEFINES variable is used instead.

```
RC_DEFINES += USE_MY_STUFF
```

Windows only. Specifies the icons that should be included into a generated .rc file. This is only utilized if the RC_FILE and RES_FILE variable are not set. More details about the generation of .rc files can be found in the Platform Notes.

RC_LANG

Windows only. Specifies the language that should be specified in a generated .rc file. This is only utilized if the VERSION or RC_ICONS variable is set and the RC_FILE and RES_FILE variables are not set.

RC_INCLUDEPATH

Specifies include paths that are passed to the Windows Resource Compiler.

RCC_DIR

Specifies the directory for Qt Resource Compiler output files.

For example:

```
unix:RCC_DIR = ../myproject/resources
win32:RCC_DIR = c:/myproject/resources
```

REQUIRES

Specifies a list of values that are evaluated as conditions. If any of the conditions is false, qmake skips this project (and its SUBDIRS) when building.

> **Note:** We recommend using the requires() function instead if you want to skip projects or subprojects when building.

RESOURCES

Specifies the name of the resource collection files (qrc) for the target. For more information about the resource collection file, see The Qt Resource System.

RES_FILE

Windows only. Specifies the name of the Windows resource compiler's output file for this target. See RC_FILE and Adding Windows Resource Files.

The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

SOURCES

For example:

```
SOURCES = myclass.cpp \
      login.cpp \
      mainwindow.cpp
```

See also HEADERS.

SUBDIRS

This variable, when used with the `subdirs` template specifies the names of all subdirectories or project files that contain parts of the project that need to be built. Each subdirectory specified using this variable must contain its own project file.

It is recommended that the project file in each subdirectory has the same base name as the subdirectory itself, because that makes it possible to omit the file name. For example, if the subdirectory is called `myapp`, the project file in that directory should be called `myapp.pro`.

Alternatively, you can specify a relative path to a .pro file in any directory. It is strongly recommended that you specify only paths in the current project's parent directory or its subdirectories.

For example:

```
SUBDIRS = kernel \
          tools \
          myapp
```

If you need to ensure that the subdirectories are built in a particular order, use the `.depends` modifier on the relevant SUBDIRS elements.

For example:

```
SUBDIRS += my_executable my_library tests doc
my_executable.depends = my_library
tests.depends = my_executable
```

The configuration above ensures that `my_library` is built before `my_executable` and that `my_executable` is built before `tests`. However, `doc` can be built in parallel with the other subdirectories, thus speeding up the build process.

> **Note:** Multiple dependencies can be listed and they will all be built before the target that depends on them.

> **Note:** Using CONFIG += ordered is discouraged as it can slow down multi-core builds. Unlike the example shown above, all builds will happen sequentially even if they don't have dependencies.

modifiers to SUBDIRS elements. Supported modifiers are:

Modifier	Effect
.subdir	Use the specified subdirectory instead of SUBDIRS value.
.file	Specify the subproject `pro` file explicitly. Cannot be used in conjunction with `.subdir` modifier.
.depends	This subproject depends on specified subproject(s).
.makefile	The makefile of subproject. Available only on platforms that use makefiles.
.target	Base string used for makefile targets related to this subproject. Available only on platforms that use makefiles.

For example, define two subdirectories, both of which reside in a different directory than the SUBDIRS value, and one of the subdirectories must be built before the other:

```
SUBDIRS += my_executable my_library
my_executable.subdir = app
my_executable.depends = my_library
my_library.subdir = lib
```

TARGET

Specifies the name of the target file. Contains the base name of the project file by default.

For example:

```
TEMPLATE = app
TARGET = myapp
SOURCES = main.cpp
```

The project file above would produce an executable named `myapp` on unix and `myapp.exe` on Windows.

TARGET_EXT

Specifies the extension of TARGET. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

TARGET_x

Specifies the extension of TARGET with a major version number. The value of this variable is typically handled by qmake or qmake.conf and rarely needs to be modified.

TARGET_x.y.z

or qmake.conf and rarely needs to be modified.

TEMPLATE

Specifies the name of the template to use when generating the project. The allowed values are:

Option	Description
app	Creates a Makefile for building applications (the default). See Building an Application for more information.
lib	Creates a Makefile for building libraries. See Building a Library for more information.
subdirs	Creates a Makefile for building targets in subdirectories. The subdirectories are specified using the SUBDIRS variable.
aux	Creates a Makefile for not building anything. Use this if no compiler needs to be invoked to create the target; for instance, because your project is written in an interpreted language. **Note:** This template type is only available for Makefile-based generators. In particular, it will not work with the vcxproj and Xcode generators.
vcapp	Windows only. Creates an application project for Visual Studio. See Creating Visual Studio Project Files for more information.
vclib	Windows only. Creates a library project for Visual Studio.

For example:

```
TEMPLATE = lib
SOURCES = main.cpp
TARGET = mylib
```

The template can be overridden by specifying a new template type with the `-t` command line option. This overrides the template type *after* the .pro file has been processed. With .pro files that use the template type to determine how the project is built, it is necessary to declare TEMPLATE on the command line rather than use the `-t` option.

TRANSLATIONS

Specifies a list of translation (.ts) files that contain translations of the user interface text into non-native languages.

Translation files in TRANSLATIONS will be processed by both lrelease and Using lupdate tools. Use EXTRA_TRANSLATIONS if you want only `lrelease` to process a file.

You can use CONFIG += lrelease to automatically compile the files during the build, and CONFIG += lrelease embed_translations to make them available in The Qt Resource System.

See the Qt Linguist Manual for more information about internationalization (i18n) and localization (l10n) with Qt.

Specifies the directory where all intermediate files from uic should be placed.

For example:

```
unix:UI_DIR = ../myproject/ui
win32:UI_DIR = c:/myproject/ui
```

VERSION

Specifies the version number of the application if the app template is specified or the version number of the library if the `lib` template is specified.

On Windows, triggers auto-generation of an .rc file if the RC_FILE and RES_FILE variables are not set. The generated .rc file will have the FILEVERSION and PRODUCTVERSION entries filled with major, minor, patch level, and build number. Each number must be in the range from 0 to 65535. More details about the generation of .rc files can be found in the Platform Notes.

For example:

```
win32:VERSION = 1.2.3.4 # major.minor.patch.build
else:VERSION = 1.2.3    # major.minor.patch
```

VERSION_PE_HEADER

Windows only. Specifies the version number, that the Windows linker puts into the header of the .exe or .dll file via the /VERSION option. Only a major and minor version may be specified. If VERSION_PE_HEADER is not set, it falls back to the major and minor version from VERSION (if set).

```
VERSION_PE_HEADER = 1.2
```

VER_MAJ

Specifies the major version number of the library if the `lib` template is specified.

VER_MIN

Specifies the minor version number of the library if the `lib` template is specified.

VER_PAT

Specifies the patch version number of the library if the `lib` template is specified.

Tells qmake where to search for files it cannot open. For example, if qmake looks for SOURCES and finds an entry that it cannot open, it looks through the entire VPATH list to see if it can find the file on its own.

See also DEPENDPATH.

WINDOWS_TARGET_PLATFORM_VERSION

Specifies the targeted Windows version; this corresponds to the tag `WindowsTargetPlatformVersion` in vcxproj files.

On desktop Windows, the default value is the value of the environment variable `WindowsSDKVersion`.

WINDOWS_TARGET_PLATFORM_MIN_VERSION

Specifies the minimum version of the Windows target platform; this corresponds to the tag `WindowsTargetPlatformMinVersion` in vcxproj files.

Defaults to `WINDOWS_TARGET_PLATFORM_VERSION`.

YACCSOURCES

Specifies a list of Yacc source files to be included in the project. All dependencies, headers and source files will automatically be included in the project.

For example:

```
YACCSOURCES = moc.y
```

_PRO_FILE_

Contains the path to the project file in use.

For example, the following line causes the location of the project file to be written to the console:

```
message($$_PRO_FILE_)
```

Note: Do not attempt to overwrite the value of this variable.

_PRO_FILE_PWD_

Contains the path to the directory containing the project file in use.

```
message($$_PRO_FILE_PWD_)
```

Note: Do not attempt to overwrite the value of this variable.

See also QQmlEngine::addImportPath().

‹ Reference

Replace Functions ›