

# Qt Documentation

Google Search Documentation



Qt 5.11 > qmake Manual

## Contents



[Table of Contents](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)

[Qt Creator Manual](#)

[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)

[What's New in Qt 5](#)

[Examples and Tutorials](#)

Supported Platforms

Qt Licensing

Overviews

Development Tools

User Interfaces

Core Internals

Data Storage

Multimedia

Networking and Connectivity

Graphics

Mobile APIs

QML Applications

All Qt Overviews

## qmake Manual

The qmake tool helps simplify the build process for development projects across different platforms. It automates the generation of Makefiles so that only a few lines of information are needed to create each Makefile. You can use qmake for any software project, whether it is written with Qt or not.

qmake generates a Makefile based on the information in a project file. Project files are created by the developer, and are usually simple, but more sophisticated project files can be created for complex projects.

qmake contains additional features to support development with Qt, automatically including build rules for [moc](#) and [uic](#).

qmake can also generate projects for Microsoft Visual studio without requiring the developer to change the project file.

# Table of Contents

- › [Overview](#)
- › [Getting Started](#)
- › [Creating Project Files](#)
- › [Building Common Project Types](#)
- › [Running qmake](#)
- › [Platform Notes](#)
- › [qmake Language](#)
- › [Advanced Usage](#)
- › [Using Precompiled Headers](#)
- › [Configuring qmake](#)
- › [Reference](#)
  - › [Variables](#)
  - › [Replace Functions](#)
    - › [Built-in Replace Functions](#)
  - › [Test Functions](#)
    - › [Built-in Test Functions](#)
    - › [Test Function Library](#)

[Overview >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > Overview

## Contents



[Describing a Project](#)  
[Building a Project](#)  
[Using Third Party Libraries](#)  
[Precompiling Headers](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)  
[All QML Types](#)  
[All Qt Modules](#)  
[Qt Creator Manual](#)  
[All Qt Reference Documentation](#)

## Getting Started

Getting Started with Qt

What's New in Qt 5

Examples and Tutorials

Supported Platforms

Qt Licensing

Overviews

Development Tools

User Interfaces

Core Internals

Data Storage

Multimedia

Networking and Connectivity

Graphics

Mobile APIs

QML Applications

All Qt Overviews

## Overview

The qmake tool provides you with a project-oriented system for managing the build process for applications, libraries, and other components. This approach gives you control over the source files used, and allows each of the steps in the process to be described concisely, typically within a single file. qmake expands the information in each project file to a Makefile that executes the necessary commands for compiling and linking.

# Describing a Project

Projects are described by the contents of project (.pro) files. **qmake uses the information within the files to generate Makefiles that contain all the commands that are needed to build each project. Project files typically contain a list of source and header files, general configuration information, and any application-specific details, such as a list of extra libraries to link against, or a list of extra include paths to use.**

Project files can contain a number of different elements, including comments, variable declarations, built-in functions, and some simple control structures. In most simple projects, it is only necessary to declare the source and header files that are used to build the project with some basic configuration options. For more information about how to create a simple project file, see [Getting Started](#).

You can create more sophisticated project files for complex projects. For an overview of project files, see [Creating Project Files](#). For detailed information about the variables and functions that you can use in project files, see [Reference](#).

You can use application or library project templates to specify specialized configuration options to fine tune the build process. For more information, see [Building Common Project Types](#).

You can use the [Qt Creator new project wizard](#) to create the project file. You choose the project template, and Qt Creator creates a project file with default values that enable you to build and run the project. You can modify the project file to suit your purposes.

You can also use qmake to generate project files. For a full description of qmake command line options, see [Running qmake](#).

The basic configuration features of qmake can handle most cross-platform projects. However, it might be useful, or even necessary, to use some platform-specific variables. For more information, see [Platform Notes](#).

## Building a Project

**For simple projects, you only need to run qmake in the top level directory of your project to generate a Makefile. You can then run your platform's make tool to build the project according to the Makefile.**

For more information about the environment variables that qmake uses when configuring the build process, see [Configuring qmake](#).

## Using Third Party Libraries

## Using Third Party Libraries

The guide to [Third Party Libraries](#) shows you how to use simple third party libraries in your Qt project.

## Precompiling Headers

In large projects, it is possible to take advantage of precompiled header files to speed up the build process. For more information, see [Using Precompiled Headers](#).

[< qmake Manual](#)

[Getting Started >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

### Download

Start for Free  
Qt for Application Development  
Qt for Device Creation  
Qt Open Source  
Terms & Conditions  
Licensing FAQ

### Product

Qt in Use  
Qt for Application Development  
Qt for Device Creation  
Commercial Features  
Qt Creator IDE  
Qt Quick

### Services

Technology Evaluation  
Proof of Concept  
Design & Implementation  
Productization  
Qt Training  
Partner Network

### Developers

Documentation  
Examples & Tutorials  
Development Tools  
Wiki  
Forums  
Contribute to Qt

### About us

Training & Events  
Resource Center  
News  
Careers  
Locations  
Contact Us







# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Getting Started](#)

## Contents



- [Starting Off Simple](#)
- [Making an Application Debuggable](#)
- [Adding Platform-Specific Source Files](#)
- [Stopping qmake If a File Does Not Exist](#)
- [Checking for More than One Condition](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



- [All Qt C++ Classes](#)
- [All QML Types](#)
- [All Qt Modules](#)
- [Qt Creator Manual](#)
- [All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)

[What's New in Qt 5](#)

[Examples and Tutorials](#)

[Supported Platforms](#)

[Qt Licensing](#)

## Overviews

[Development Tools](#)

[User Interfaces](#)

[Core Internals](#)

[Data Storage](#)

[Multimedia](#)

[Networking and Connectivity](#)

[Graphics](#)

[Mobile APIs](#)

[QML Applications](#)

[All Qt Overviews](#)

# Getting Started

This tutorial teaches you the basics of qmake. The other topics in this manual contain more detailed information about using qmake.

# Starting Off Simple

Let's assume that you have just finished a basic implementation of your application, and you have created the following files:

- › hello.cpp
- › hello.h
- › main.cpp

You will find these files in the `examples/qmake/tutorial` directory of the Qt distribution. The only other thing you know about the setup of the application is that it's written in Qt. First, using your favorite plain text editor, create a file called `hello.pro` in `examples/qmake/tutorial`. The first thing you need to do is add the lines that tell qmake about the source and header files that are part of your development project.

We'll add the source files to the project file first. To do this you need to use the `SOURCES` variable. Just start a new line with `SOURCES +=` and put `hello.cpp` after it. You should have something like this:

```
SOURCES += hello.cpp
```

添加源文件



We repeat this for each source file in the project, until we end up with the following:

```
SOURCES += hello.cpp  
SOURCES += main.cpp
```

If you prefer to use a Make-like syntax, with all the files listed in one go you can use the newline escaping like this:

```
SOURCES = hello.cpp \  
          main.cpp
```

Now that the source files are listed in the project file, the header files must be added. These are added in exactly the same way as source files, except that the variable name we use is `HEADERS`.

Once you have done this, your project file should look something like this:

```
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp
```



添加头文件

The target name is set automatically. It is the same as the project filename, but with the suffix appropriate for the platform. For example, if the project file is called `hello.pro`, the target will be `hello.exe` on Windows and `hello` on Unix. If you want to use a different name you can set it in the project file:

```
TARGET = helloworld
```



生成的对象的名字

The finished project file should look like this:

```
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp
```

You can now use `qmake` to generate a Makefile for your application. On the command line, in your project directory, type the following:

```
qmake -o Makefile hello.pro
```



生成Makefile文件

Then type make or nmake depending on the compiler you use.

For Visual Studio users, qmake can also generate Visual Studio project files. For example:

```
qmake -tp vc hello.pro
```

## Making an Application Debuggable

The release version of an application does not contain any debugging symbols or other debugging information. During development, it is useful to produce a debugging version of the application that has the relevant information. This is easily achieved by adding debug to the CONFIG variable in the project file.

For example:

```
CONFIG += debug  
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp
```

生成调试版本



Use qmake as before to generate a Makefile. You will now obtain useful information about your application when running it in a debugging environment.

## Adding Platform-Specific Source Files

After a few hours of coding, you might have made a start on the platform-specific part of your application, and decided to keep the platform-dependent code separate. So you now have two new files to include into your project file: helloworldin.cpp and helloworldunix.cpp. We cannot just

add these to the `SOURCES` variable since that would place both files in the Makefile. So, what we need to do here is to use a scope which will be processed depending on which platform we are building for.

A simple scope that adds the platform-dependent file for Windows looks like this:

```
win32 {  
    SOURCES += helloworld.cpp  
}
```

添加平台相关的文件的方式

When building for Windows, qmake adds `helloworld.cpp` to the list of source files. When building for any other platform, qmake simply ignores it. Now all that is left to be done is to create a scope for the Unix-specific file.

When you have done that, your project file should look something like this:

```
CONFIG += debug  
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp  
win32 {  
    SOURCES += helloworld.cpp  
}  
unix {  
    SOURCES += hellounix.cpp  
}
```

类似于条件检测

Use qmake as before to generate a Makefile.

## Stopping qmake If a File Does Not Exist

You may not want to create a Makefile if a certain file does not exist. We can check if a file exists by using the `exists()` function. We can stop qmake from processing by using the `error()` function. This works in the same way as scopes do. Simply replace the scope condition with the function. A check for a file called `main.cpp` looks like this:

```
!exists( main.cpp ) {  
    error( "No main.cpp file found" )  
}
```

用 `exists()` 检测文件的存在，用 `error()` 阻止make的继续执行

The `!` symbol is used to negate the test. That is, `exists( main.cpp )` is true if the file exists, and `!exists( main.cpp )` is true if the file does not exist.

```
CONFIG += debug  
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp  
win32 {  
    SOURCES += hellowin.cpp  
}  
unix {  
    SOURCES += hellounix.cpp  
}  
!exists( main.cpp ) {  
    error( "No main.cpp file found" )  
}
```

Use qmake as before to generate a makefile. If you rename `main.cpp` temporarily, you will see the message and qmake will stop processing.

## Checking for More than One Condition



Suppose you use Windows and you want to be able to see statement output with `qDebug()` when you run your application on the command line. To see the output, you must build your application with the appropriate console setting. We can easily put `console` on the `CONFIG` line to include this setting in the Makefile on Windows. However, let's say that we only want to add the `CONFIG` line when we are running on Windows *and* when `debug` is already on the `CONFIG` line. This requires using two nested scopes. First create one scope, then create the other inside it. Put the settings to be processed inside the second scope, like this:

```
win32 {  
    debug {  
        CONFIG += console  
    }  
}
```

多重条件检测

Nested scopes can be joined together using colons, so the final project file looks like this:

```
CONFIG += debug  
HEADERS += hello.h  
SOURCES += hello.cpp  
SOURCES += main.cpp  
win32 {  
    SOURCES += helloworld.cpp  
}  
unix {  
    SOURCES += helloworld.cpp  
}  
!exists( main.cpp ) {  
    error( "No main.cpp file found" )  
}  
win32:debug {  
    CONFIG += console  
}
```

多重条件检测可以用:连接

That's it! You have now completed the tutorial for qmake, and are ready to write project files for your development projects.

[< Overview](#)

[Creating Project Files >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

#### Download

Start for Free  
Qt for Application Development  
Qt for Device Creation  
Qt Open Source  
Terms & Conditions  
Licensing FAQ

#### Product

Qt in Use  
Qt for Application Development  
Qt for Device Creation  
Commercial Features  
Qt Creator IDE  
Qt Quick

#### Services

Technology Evaluation  
Proof of Concept  
Design & Implementation  
Productization  
Qt Training  
Partner Network

#### Developers

Documentation  
Examples & Tutorials  
Development Tools  
Wiki  
Forums  
Contribute to Qt

#### About us

Training & Events  
Resource Center  
News  
Careers  
Locations  
Contact Us



[Sign In](#) [Feedback](#) © 2018 The Qt Company



# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Creating Project Files](#)

## Contents



[Project File Elements](#)

[Variables](#)

[Comments](#)

[Built-in Functions and Control Flow](#)

[Project Templates](#)

[General Configuration](#)

[Declaring Qt Libraries](#)

[Configuration Features](#)

[Declaring Other Libraries](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)  
[Qt Creator Manual](#)  
[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)  
[What's New in Qt 5](#)  
[Examples and Tutorials](#)  
[Supported Platforms](#)  
[Qt Licensing](#)

## Overviews

[Development Tools](#)  
[User Interfaces](#)  
[Core Internals](#)  
[Data Storage](#)  
[Multimedia](#)  
[Networking and Connectivity](#)  
[Graphics](#)  
[Mobile APIs](#)  
[QML Applications](#)  
[All Qt Overviews](#)

# Creating Project Files

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

## Project File Elements

The project file format used by qmake can be used to support both simple and fairly complex build systems. Simple project files use a straightforward declarative style, defining standard variables to indicate the source and header files that are used in the project. Complex projects may use control flow structures to fine-tune the build process.

The following sections describe the different types of elements used in project files.

### Variables

变量的作用



In a project file, variables are used to hold lists of strings. In the simplest projects, these variables inform qmake about the configuration options to use, or supply filenames and paths to use in the build process.

qmake looks for certain variables in each project file, and it uses the contents of these to determine what it should write to a Makefile. For example, the lists of values in the **HEADERS** and **SOURCES** variables are used to tell qmake about header and source files in the same directory as the project file.

Variables can also be used internally to store temporary lists of values, and existing lists of values can be overwritten or extended with new values.

The following snippet illustrates how lists of values are assigned to variables:

```
HEADERS = mainwindow.h paintwidget.h
```

The list of values in a variable is extended in the following way:

```
SOURCES = main.cpp mainwindow.cpp \  
        paintwidget.cpp  
CONFIG += console
```

**Note:** The first assignment only includes values that are specified on the same line as the `HEADERS` variable. The second assignment splits the values in the `SOURCES` variable across lines by using a backslash (`\`).

The `CONFIG` variable is another special variable that `qmake` uses when generating a Makefile. It is discussed in [General Configuration](#). In the snippet above, `console` is added to the list of existing values contained in `CONFIG`.

The following table lists some frequently used variables and describes their contents. For a full list of variables and their descriptions, see [Variables](#).

Variable	Contents
<code>CONFIG</code>	General project configuration options.
<code>DESTDIR</code>	The directory in which the executable or binary file will be placed.
<code>FORMS</code>	A list of UI files to be processed by the <a href="#">user interface compiler (uic)</a> .
<code>HEADERS</code>	A list of filenames of header ( <code>.h</code> ) files used when building the project.
<code>QT</code>	A list of Qt modules used in the project.
<code>RESOURCES</code>	A list of resource ( <code>.qrc</code> ) files to be included in the final project. See the <a href="#">The Qt Resource System</a> for more information about these files.
<code>SOURCES</code>	A list of source code files to be used when building the project.
<code>TEMPLATE</code>	The template to use for the project. This determines whether the output of the build process will be an application, a library, or a plugin.

The contents of a variable can be read by prepending the variable name with `$$`. This can be used to assign the contents of one variable to another:

```
TEMP_SOURCES = $$SOURCES
```

使用\$\$对已经定义的变量进行引用

The `$$` operator is used extensively with built-in functions that operate on strings and lists of values. For more information, see [qmake Language](#).

## Whitespace

↑ `$$`也可以对内建函数进行引用

Usually, whitespace separates values in variable assignments. To specify values that contain spaces, you must enclose the values in double quotes:

```
DEST = "Program Files"
```

The quoted text is treated as a single item in the list of values held by the variable. A similar approach is used to deal with paths that contain spaces, particularly when defining the `INCLUDEPATH` and `LIBS` variables for the Windows platform:

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

← 变量值内部的空格的处理方式

## Comments

You can add comments to project files. Comments begin with the `#` character and continue to the end of the same line. For example:

```
# Comments usually start at the beginning of a line, but they  
# can also follow other content on the same line.
```

← 注释方式

To include the `#` character in variable assignments, it is necessary to use the contents of the built-in `LITERAL_HASH` variable.

## Built-in Functions and Control Flow



qmake provides a number of built-in functions to enable the contents of variables to be processed. The most commonly used function in simple project files is the `include()` function which takes a filename as an argument. The contents of the given file are included in the project file at the place where the `include` function is used. The `include` function is most commonly used to include other project files:

```
include(other.pro)
```

## `include()` 函数的使用

Support for conditional structures is made available via `scopes` that behave like `if` statements in programming languages:

```
win32 {  
    SOURCES += paintwidget_win.cpp  
}
```

The assignments inside the braces are only made if the condition is true. In this case, the `win32 CONFIG` option must be set. This happens automatically on Windows. The opening brace must stand on the same line as the condition.

More complex operations on variables that would usually require loops are provided by built-in functions such as `find()`, `unique()`, and `count()`. These functions, and many others are provided to manipulate strings and paths, support user input, and call external tools. For more information about using the functions, see [qmake Language](#). For lists of all functions and their descriptions, see [Replace Functions](#) and [Test Functions](#).

## Project Templates

The `TEMPLATE` variable is used to define the type of project that will be built. If this is not declared in the project file, qmake assumes that an application should be built, and will generate an appropriate Makefile (or equivalent file) for the purpose.

The following table summarizes the types of projects available and describes the files that qmake will generate for each of them:

Template	qmake Output
app	Makefile to build an application

app (default)	Makefile to build an application.
lib	Makefile to build a library.
aux	Makefile to build nothing. Use this if no compiler needs to be invoked to create the target, for instance because your project is written in an interpreted language. <b>Note:</b> This template type is only available for Makefile-based generators. In particular, it will not work with the vcxproj and Xcode generators.
subdirs	Makefile containing rules for the subdirectories specified using the SUBDIRS variable. Each subdirectory must contain its own project file.
vcapp	Visual Studio Project file to build an application.
vclib	Visual Studio Project file to build a library.
vcsubdirs	Visual Studio Solution file to build projects in sub-directories.

See [Building Common Project Types](#) for advice on writing project files for projects that use the `app` and `lib` templates.

When the `subdirs` template is used, `qmake` generates a Makefile to examine each specified subdirectory, process any project file it finds there, and run the platform's make tool on the newly-created Makefile. The `SUBDIRS` variable is used to contain a list of all the subdirectories to be processed.

## General Configuration

The `CONFIG` variable specifies the options and features that the project should be configured with.

The project can be built in *release* mode or *debug* mode, or both. If debug and release are both specified, the last one takes effect. If you specify the `debug_and_release` option to build both the debug and release versions of a project, the Makefile that `qmake` generates includes a rule that builds both versions. This can be invoked in the following way:

```
make all
```

注释版本和调试版本的控制方式

Adding the `build_all` option to the `CONFIG` variable makes this rule the default when building the project.

**Note:** Each of the options specified in the `CONFIG` variable can also be used as a scope condition. You can test for the presence of certain configuration options by using the built-in `CONFIG()` function. For example, the following lines show the function as the condition in a scope to test whether only the `opengl` option is in use:

```
CONFIG(opengl) {  
    message(Building with OpenGL support.)  
} else {  
    message(OpenGL support is not available.)  
}
```

This enables different configurations to be defined for `release` and `debug` builds. For more information, see [Using Scopes](#).

The following options define the type of project to be built.

**Note:** Some of these options only take effect when used on the relevant platform.

Option	Description
qt	The project is a Qt application and should link against the Qt library. You can use the <code>QT</code> variable to control any additional Qt modules that are required by your application. This value is added by default, but you can remove it to use <code>qmake</code> for a non-Qt project.
x11	The project is an X11 application or library. This value is not needed if the target uses Qt.

The [application and library project templates](#) provide you with more specialized configuration options to fine tune the build process. The options are explained in detail in [Building Common Project Types](#).

For example, if your application uses the Qt library and you want to build it in `debug` mode, your project file will contain the following line:

```
CONFIG += qt debug
```

**Note:** You must use "+=", not "=", or qmake will not be able to use Qt's configuration to determine the settings needed for your project.

## Declaring Qt Libraries

If the `CONFIG` variable contains the `qt` value, qmake's support for Qt applications is enabled. This makes it possible to fine-tune which of the Qt modules are used by your application. This is achieved with the `QT` variable which can be used to declare the required extension modules. For example, we can enable the XML and network modules in the following way:

```
QT += network xml
```

← 添加扩展的模块，默认值为core和gui 模块

**Note:** QT includes the `core` and `gui` modules by default, so the above declaration *adds* the network and XML modules to this default list. The following assignment *omits* the default modules, and will lead to errors when the application's source code is being compiled:

```
QT = network xml # This will omit the core and gui modules.
```

If you want to build a project *without* the `gui` module, you need to exclude it with the `"-="` operator. By default, QT contains both `core` and `gui`, so the following line will result in a minimal Qt project being built:

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

For a list of Qt modules that you can add to the QT variable, see [QT](#).

## Configuration Features

qmake can be set up with extra configuration features that are specified in feature (.prf) files. These extra features often provide support for custom tools that are used during the build process. To add a feature to the build process, append the feature name (the stem of the feature filename) to the **CONFIG** variable.

For example, qmake can configure the build process to take advantage of external libraries that are supported by **pkg-config**, such as the D-Bus and ogg libraries, with the following lines:

```
CONFIG += link_pkgconfig  
PKGCONFIG += ogg dbus-1
```

这个注意下如何使用

For more information about adding features, see [Adding New Configuration Features](#).

## Declaring Other Libraries

If you are using other libraries in your project in addition to those supplied with Qt, you need to specify them in your project file.

The paths that qmake searches for libraries and the specific libraries to link against can be added to the list of values in the **LIBS** variable. You can specify the paths to the libraries or use the Unix-style notation for specifying libraries and paths.

For example, the following lines show how a library can be specified:

```
LIBS += -L/usr/local/lib -lmath
```

使用其它的库的方式，使用LIBS追加。

The paths containing header files can also be specified in a similar way using the **INCLUDEPATH** variable.

For example, to add several paths to be searched for header files:

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

[< Getting Started](#)

## 头文件的处理方式，使用INCLUDEPATH

[Building Common Project Types >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

### Download

- [Start for Free](#)
- [Qt for Application Development](#)
- [Qt for Device Creation](#)
- [Qt Open Source](#)
- [Terms & Conditions](#)
- [Licensing FAQ](#)

### Product

- [Qt in Use](#)
- [Qt for Application Development](#)
- [Qt for Device Creation](#)
- [Commercial Features](#)
- [Qt Creator IDE](#)
- [Qt Quick](#)

### Services

- [Technology Evaluation](#)
- [Proof of Concept](#)
- [Design & Implementation](#)
- [Productization](#)
- [Qt Training](#)
- [Partner Network](#)

### Developers

- [Documentation](#)
- [Examples & Tutorials](#)
- [Development Tools](#)
- [Wiki](#)
- [Forums](#)
- [Contribute to Qt](#)

### About us

- [Training & Events](#)
- [Resource Center](#)
- [News](#)
- [Careers](#)
- [Locations](#)
- [Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company



# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > Building Common Project Types

## Contents



[Building an Application](#)

[Building a Testcase](#)

[Building a Library](#)

[Building a Plugin](#)

[Building a Qt Designer Plugin](#)

[Building and Installing in Debug and Release Modes](#)

[Building in Both Modes](#)

[Installing in Both Modes](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)



Qt Creator Manual  
All Qt Reference Documentation

## Getting Started

Getting Started with Qt  
What's New in Qt 5  
Examples and Tutorials  
Supported Platforms  
Qt Licensing

## Overviews

Development Tools  
User Interfaces  
Core Internals  
Data Storage  
Multimedia  
Networking and Connectivity  
Graphics  
Mobile APIs  
QML Applications  
All Qt Overviews

# Building Common Project Types

This chapter describes how to set up qmake project files for three common project types that are based on Qt: application, library, and plugin. Although all project types use many of the same variables, each of them uses project-specific variables to customize output files.

Platform-specific variables are not described here. For more information, see [Qt for Windows - Deployment and Qt for macOS](#).

## 三种常见项目的配置方式

### Building an Application

The `app` template tells qmake to generate a Makefile that will build an application. With this template, the type of application can be specified by adding one of the following options to the `CONFIG` variable definition:

Option	Description
windows	The application is a Windows GUI application.
console	app template only: the application is a Windows console application.
testcase	The application is an automated test.

## 通过CONFIG进行配置

When using this template, the following qmake system variables are recognized. You should use these in your `.pro` file to specify information about your application. For additional platform-dependent system variables, you could have a look at the [Platform Notes](#).

- › `HEADERS` - A list of header files for the application.
- › `SOURCES` - A list of C++ source files for the application.
- › `FORMS` - A list of UI files for the application (created using Qt Designer).
- › `LEXSOURCES` - A list of Lex source files for the application.
- › `YACCSOURCES` - A list of Yacc source files for the application.
- › `TARGET` - Name of the executable for the application. This defaults to the name of the project file. (The extension, if any, is added automatically).
- › `DESTDIR` - The directory in which the target executable is placed.
- › `DEFINES` - A list of any additional pre-processor defines needed for the application.
- › `INCLUDEPATH` - A list of any additional include paths needed for the application.
- › `DEPENDPATH` - The dependency search path for the application.
- › `VPATH` - The search path to find supplied files.

- › `DEF_FILE` - Windows only: A .def file to be linked against for the application.

You only need to use the system variables that you have values for. For example, if you do not have any extra `INCLUDEPATHs` then you do not need to specify any. qmake will add the necessary default values. An example project file might look like this:

```
TEMPLATE = app
DESTDIR  = c:/helloapp
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
DEFINES += USE_MY_STUFF
CONFIG += release
```

有缺省的头文件路径

For items that are single valued, such as the template or the destination directory, we use "="; but for multi-valued items we use "+=" to *add* to the existing items of that type. Using "=" replaces the variable value with the new value. For example, if we write `DEFINES=USE_MY_STUFF`, all other definitions are deleted.

## Building a Testcase

A testcase project is an app project intended to be run as an automated test. Any app may be marked as a testcase by adding the value `testcase` to the `CONFIG` variable.

For testcase projects, qmake will insert a `check` target into the generated Makefile. This target will run the application. The test is considered to pass if it terminates with an exit code equal to zero.

The `check` target automatically recurses through `SUBDIRS` projects. This means it is possible to issue a `make check` command from within a `SUBDIRS` project to run an entire test suite.

The execution of the `check` target may be customized by certain Makefile variables. These variables are:

Variable	Description
TESTRUNNER	A command or shell fragment prepended to each test command. An example use-case is a "timeout" script which will terminate a test if it does not complete within a specified time.
TESTARGS	Additional arguments appended to each test command. For example, it may be useful to pass additional arguments to set the output file and format from the test (such as the <code>-o filename</code> , format option supported by <a href="#">QTestLib</a> ).

**Note:** The variables must be set while invoking the make tool, not in the .pro file. Most make tools support the setting of Makefile variables directly on the command-line:

```
# Run tests through test-wrapper and use xunitxml output format.
# In this example, test-wrapper is a fictional wrapper script which terminates
# a test if it does not complete within the amount of seconds set by "--timeout".
# The "-o result.xml,xunitxml" options are interpreted by QTestLib.
make check TESTRUNNER="test-wrapper --timeout 120" TESTARGS="-o result.xml,xunitxml"
```

Testcase projects may be further customized with the following CONFIG options:

Option	Description
insignificant_test	The exit code of the test will be ignored during <code>make check</code> .

Testcases will often be written with [QTest](#) or [TestCase](#), but that is not a requirement to make use of `CONFIG+=testcase` and `make check`. The only primary requirement is that the test program exit with a zero exit code on success, and a non-zero exit code on failure.

## Building a Library

生成动态库

版本的控制

The `lib` template tells `qmake` to generate a Makefile that will build a library. When using this template, the `VERSION` variable is supported, in addition to the system variables that the `app` template supports. Use the variables in your .pro file to specify information about the library.

When using the `lib` template, the following options can be added to the `CONFIG` variable to determine the type of library that is built:

Option	Description
dll	The library is a shared library (dll).
staticlib	The library is a static library.
plugin	The library is a plugin.

The following option can also be defined to provide additional information about the library.

- › **VERSION** - The version number of the target library. For example, 2.3.1.

版本version的控制

The target file name for the library is platform-dependent. For example, on X11, macOS, and iOS, the library name will be prefixed by `lib`. On Windows, no prefix is added to the file name.

## Building a Plugin

插件也是动态库

Plugins are built using the `lib` template, as described in the previous section. This tells qmake to generate a Makefile for the project that will build a plugin in a suitable form for each platform, usually in the form of a library. As with ordinary libraries, the `VERSION` variable is used to specify information about the plugin.

- › **VERSION** - The version number of the target library. For example, 2.3.1.

## Building a Qt Designer Plugin

*Qt Designer* plugins are built using a specific set of configuration settings that depend on the way Qt was configured for your system. For convenience, these settings can be enabled by adding `designer` to the `QT` variable. For example:

```
QT += widgets designer
```

See the [Qt Designer Examples](#) for more examples of plugin-based projects.

# Building and Installing in Debug and Release Modes

Sometimes, it is necessary to build a project in both debug and release modes. Although the `CONFIG` variable can hold both debug and release options, only the option that is specified last is applied.

## Building in Both Modes

To enable a project to be built in both modes, you must add the `debug_and_release` option to the `CONFIG` variable:

```
CONFIG += debug_and_release

CONFIG(debug, debug|release) {
    TARGET = debug_binary
} else {
    TARGET = release_binary
}
```

测试结果生成的是debug版本



The scope in the above snippet modifies the build target in each mode to ensure that the resulting targets have different names. Providing different names for targets ensures that one will not overwrite the other.

When qmake processes the project file, it will generate a Makefile rule to allow the project to be built in both modes. This can be invoked in the following way:

```
make all
```

The `build_all` option can be added to the `CONFIG` variable in the project file to ensure that the project is built in both modes by default:

```
CONFIG += build_all
```

This allows the Makefile to be processed using the default rule:

```
make
```

## Installing in Both Modes

The `build_all` option also ensures that both versions of the target will be installed when the installation rule is invoked:

```
make install
```

It is possible to customize the names of the build targets depending on the target platform. For example, a library or plugin may be named using a different convention on Windows from the one used on Unix platforms:

```
CONFIG(debug, debug|release) {  
    mac: TARGET = $$join(TARGET,,,_debug)  
    win32: TARGET = $$join(TARGET,,d)  
}
```

← 调整debug版本的后缀

The default behavior in the above snippet is to modify the name used for the build target when building in debug mode. An `else` clause could be added to the scope to do the same for release mode. Left as it is, the target name remains unmodified.

[< Creating Project Files](#)

[Running qmake >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company



# Qt Documentation

Google Custom Search



Qt 5.11 > [qmake Manual](#) > Running qmake

## Contents



[Command Syntax](#)

[Operating Modes](#)

[Files](#)

[General Options](#)

[Makefile Mode Options](#)

[Project Mode Options](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)

[Qt Creator Manual](#)

[All Qt Reference Documentation](#)

Getting Started

[Getting Started with Qt](#)

[What's New in Qt 5](#)

[Examples and Tutorials](#)

[Supported Platforms](#)

[Qt Licensing](#)

Overviews

[Development Tools](#)

[User Interfaces](#)

[Core Internals](#)

[Data Storage](#)

[Multimedia](#)

[Networking and Connectivity](#)

[Graphics](#)

[Mobile APIs](#)

[QML Applications](#)

[All Qt Overviews](#)

## Running qmake

The behavior of qmake can be customized when it is run by specifying various options on the command line. These allow the build process to be fine-tuned, provide useful diagnostic information, and can be used to specify the target platform for your project.

tuned, provide useful diagnostic information, and can be used to specify the target platform for your project.

## Command Syntax

The syntax used to run qmake takes the following simple form:

```
qmake [mode] [options] files
```

## Operating Modes

qmake supports two different modes of operation. In the default mode, qmake uses the information in a project file to generate a Makefile, but it is also possible to use qmake to generate project files. If you want to explicitly set the mode, you must specify it before all other options. The mode can be either of the following two values:

- › -makefile  
qmake output will be a Makefile.
- › -project  
qmake output will be a project file.

两种模式的选择



**Note:** It is likely that the created file will need to be edited. For example, adding the QT variable to suit what modules are required for the project.

You can use the options to specify both general and mode-specific settings. Options that only apply to the Makefile mode are described in the [Makefile Mode Options](#) section, whereas options that influence the creation of project files are described in the [Project Mode Options](#) section.

## Files

The `files` argument represents a list of one or more project files, separated by spaces.

# General Options

A wide range of options can be specified on the command line to qmake in order to customize the build process, and to override default settings for your platform. The following basic options provide help on using qmake, specify where qmake writes the output file, and control the level of debugging information that will be written to the console:

- › -help  
qmake will go over these features and give some useful help.
- › -o file  
qmake output will be directed to file. If this option is not specified, qmake will try to use a suitable file name for its output, depending on the mode it is running in.  
If '-' is specified, output is directed to stdout.
- › -d  
qmake will output debugging information. Adding -d more than once increases verbosity.

The template used for the project is usually specified by the `TEMPLATE` variable in the project file. You can override or modify this by using the following options:

- › -t tmpl  
qmake will override any set `TEMPLATE` variables with `tmpl`, but only *after* the `.pro` file has been processed.
- › -tp prefix  
qmake will add prefix to the `TEMPLATE` variable.

The level of warning information can be fine-tuned to help you find problems in your project file:

- › -Wall  
qmake will report all known warnings.
- › -Wnone  
No warning information will be generated by qmake.
- › -Wparser  
qmake will only generate parser warnings. This will alert you to common pitfalls and potential problems in the parsing of your project files.
- › -Wlogic  
qmake will warn of common pitfalls and potential problems in your project file. For example, qmake will report multiple occurrences of files in lists and missing files.

# Makefile Mode Options

```
qmake -makefile [options] files
```

In Makefile mode, qmake will generate a Makefile that is used to build the project. Additionally, the following options may be used in this mode to influence the way the project file is generated:

- › **-after**  
qmake will process assignments given on the command line after the specified files.
- › **-nocache**  
qmake will ignore the `.qmake.cache` file.
- › **-nodepend**  
qmake will not generate any dependency information.
- › **-cache file**  
qmake will use `file` as the cache file, ignoring any other `.qmake.cache` files found.
- › **-spec spec**  
qmake will use `spec` as a path to platform and compiler information, and ignore the value of `QMAKESPEC`.

You may also pass qmake assignments on the command line. They are processed before all of the files specified. For example, the following command generates a Makefile from `test.pro`:

```
qmake -makefile -o Makefile "CONFIG+=test" test.pro
```

However, some of the specified options can be omitted as they are default values:

```
qmake "CONFIG+=test" test.pro
```

```
qmake -conf id- test- test.pro
```

If you are certain you want your variables processed after the files specified, then you may pass the `-after` option. When this is specified, all assignments on the command line after the `-after` option will be postponed until after the specified files are parsed.

## Project Mode Options

```
qmake -project [options] files
```

In project mode, qmake will generate a project file. Additionally, you may supply the following options in this mode:

- › `-r`  
qmake will look through supplied directories recursively.
- › `-nopwd`  
qmake will not look in your current working directory for source code. It will only use the specified files.

In this mode, the `files` argument can be a list of files or directories. If a directory is specified, it will be included in the `DEPENDPATH` variable, and relevant code from there will be included in the generated project file. If a file is given, it will be appended to the correct variable, depending on its extension. For example, UI files are added to `FORMS`, and C++ files are added to `SOURCES`.

You may also pass assignments on the command line in this mode. When doing so, these assignments will be placed last in the generated project file.

[< Building Common Project Types](#)

[Platform Notes >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > Platform Notes

## Contents



[macOS, iOS, tvOS, and watchOS](#)

[Source and Binary Packages](#)

[Using Frameworks](#)

[Creating Frameworks](#)

[Creating and Moving Xcode Projects](#)

[Supporting Two Build Targets Simultaneously](#)

[Windows](#)

[Adding Windows Resource Files](#)

[Creating Visual Studio Project Files](#)

[Visual Studio Manifest Files](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)



[All QML Types](#)  
[All Qt Modules](#)  
[Qt Creator Manual](#)  
[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)  
[What's New in Qt 5](#)  
[Examples and Tutorials](#)  
[Supported Platforms](#)  
[Qt Licensing](#)

## Overviews

[Development Tools](#)  
[User Interfaces](#)  
[Core Internals](#)  
[Data Storage](#)  
[Multimedia](#)  
[Networking and Connectivity](#)  
[Graphics](#)  
[Mobile APIs](#)  
[QML Applications](#)  
[All Qt Overviews](#)

# Platform Notes

Many cross-platform projects can be handled by the basic qmake configuration features. However, on some platforms, it is sometimes useful, or even necessary, to take advantage of platform-specific features. qmake knows about many of these features, which can be accessed via specific variables that only take effect on the platforms where they are relevant.

## macOS, iOS, tvOS, and watchOS

Features specific to these platforms include support for creating universal binaries, frameworks and bundles.

### Source and Binary Packages

The version of qmake supplied in source packages is configured slightly differently to that supplied in binary packages in that it uses a different feature specification. Where the source package typically uses the `macx-g++` specification, the binary package is typically configured to use the `macx-xcode` specification.

Users of each package can override this configuration by invoking qmake with the `-spec` option (see [Running qmake](#) for more information). For example, to use qmake from a binary package to create a Makefile in a project directory, invoke the following command:

```
qmake -spec macx-g++
```

### Using Frameworks

qmake is able to automatically generate build rules for linking against frameworks in the standard framework directory on [macOS](#), located at `/Library/Frameworks/`.

Directories other than the standard framework directory need to be specified to the build system, and this is achieved by appending linker options to the `LIBS` variable, as shown in the following example:

```
LIBS += -F/path/to/framework/directory/
```

The framework itself is linked in by appending the `-framework` options and the name of the framework to the `LIBS` variable:

```
LIBS += -framework TheFramework
```

## Creating Frameworks

Any given library project can be configured so that the resulting library file is placed in a `framework`, ready for deployment. To do this, set up the project to use the `lib` template and add the `lib_bundle` option to the `CONFIG` variable:

```
TEMPLATE = lib  
CONFIG += lib_bundle
```

The data associated with the library is specified using the `QMAKE_BUNDLE_DATA` variable. This holds items that will be installed with a library bundle, and is often used to specify a collection of header files, as in the following example:

```
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
```

You use the `FRAMEWORK_HEADERS` variable to specify the headers required by a particular framework. Appending it to the `QMAKE_BUNDLE_DATA` variable ensures that information about these headers is added to the collection of resources that will be installed with the library bundle. Also, the

framework name and version are specified by the `QMAKE_FRAMEWORK_BUNDLE_NAME` and `QMAKE_FRAMEWORK_VERSION` variables. By default, the values used for these variables are obtained from the `TARGET` and `VERSION` variables.

See [Qt for macOS - Deployment](#) for more information about deploying applications and libraries.

## Creating and Moving Xcode Projects

Developers on [macOS](#) can take advantage of the qmake support for Xcode project files, as described in [Qt for macOS](#) documentation. by running qmake to generate an Xcode project from an existing qmake project file. For example:

```
qmake -spec macx-xcode project.pro
```

**Note:** If a project is later moved on the disk, qmake must be run again to process the project file and create a new Xcode project file.

## Supporting Two Build Targets Simultaneously

Implementing this is currently not feasible, because the Xcode concept of Active Build Configurations is conceptually different from the qmake idea of build targets.

The Xcode Active Build Configurations settings are for modifying Xcode configurations, compiler flags and similar build options. Unlike Visual Studio, Xcode does not allow for the selection of specific library files based on whether debug or release build configurations are selected. The qmake debug and release settings control which library files are linked to the executable.

It is currently not possible to set files in Xcode configuration settings from the qmake generated Xcode project file. The way the libraries are linked in the *Frameworks & Libraries* phase in the Xcode build system.

Furthermore, the selected *Active Build Configuration* is stored in a .pbxuser file, which is generated by Xcode on first load, not created by qmake.

## Windows

Features specific to this platform include support for windows resource files (provided or auto-generated), creating Visual Studio project files, and handling manifest files when deploying Qt applications developed using Visual Studio 2005, or later.

## Adding Windows Resource Files

This section describes how to handle a Windows resource file with qmake to have it linked to an application executable (EXE) or dynamic link library (DLL). qmake can optionally auto-generate a suitably filled Windows resource file.

A linked Windows resource file may contain many elements that can be accessed by its EXE or DLL. However, the [Qt resource system](#) should be used for accessing linked-in resources in a platform-independent way. But some standard elements of the linked Windows resource file are accessed by Windows itself. For example, in Windows explorer the version tab of the file properties is filled by resource elements. In addition, the program icon of the EXE is read from these elements. So it is good practice for a Qt created Windows EXE or DLL to use both techniques at the same time: link platform-independent resources via the [Qt resource system](#) and add Windows specific resources via a Windows resource file.

Typically, a resource-definition script (.rc file) is compiled to a Windows resource file. Within the Microsoft toolchain, the RC tool generates a .res file, which can be linked with the Microsoft linker to an EXE or DLL. The [MinGW](#) toolchain uses the windres tool to generate an .o file that can be linked with the [MinGW](#) linker to an EXE or DLL.

The optional auto-generation of a suitably filled .rc file by qmake is triggered by setting at least one of the system variables [VERSION](#) and [RC\\_ICONS](#). The generated .rc file is automatically compiled and linked. Elements that are added to the .rc file are defined by the system variables [QMAKE\\_TARGET\\_COMPANY](#), [QMAKE\\_TARGET\\_DESCRIPTION](#), [QMAKE\\_TARGET\\_COPYRIGHT](#), [QMAKE\\_TARGET\\_PRODUCT](#), [RC\\_CODEPAGE](#), [RC\\_ICONS](#), [RC\\_LANG](#), and [VERSION](#).

If these elements are not sufficient, qmake has the two system variables [RC\\_FILE](#) and [RES\\_FILE](#) that point directly to an externally created .rc or .res file. By setting one of these variables, the specified file is linked to the EXE or DLL.

**Note:** The generation of the .rc file by qmake is blocked, if [RC\\_FILE](#) or [RES\\_FILE](#) is set. In this case, no further changes are made to the given .rc file or the .res or .o file by qmake; the variables pertaining to .rc file generation have no effect.

## Creating Visual Studio Project Files

This section describes how to import an existing qmake project into Visual Studio. qmake is able to take a project file and create a Visual Studio project that contains all the necessary information required by the development environment. This is achieved by setting the qmake [project template](#) to either `vcapp` (for application projects) or `vc1ib` (for library projects).

This can also be set using a command line option, for example:

```
qmake -tp vc
```

It is possible to recursively generate `.vcproj` files in subdirectories and a `.sln` file in the main directory, by typing:

```
qmake -tp vc -r
```

Each time you update the project file, you need to run `qmake` to generate an updated Visual Studio project.

**Note:** If you are using the Visual Studio Add-in, select **Qt > Import from .pro file** to import `.pro` files.

## Visual Studio Manifest Files

When deploying Qt applications built using Visual Studio 2005, or later, make sure that the manifest file that was created when the application was linked is handled correctly. This is handled automatically for projects that generate DLLs.

Removing manifest embedding for application executables can be done with the following assignment to the `CONFIG` variable:

```
CONFIG -= embed_manifest_exe
```

Also, the manifest embedding for DLLs can be removed with the following assignment to the `CONFIG` variable:

```
CONFIG -= embed_manifest_dll
```

This is discussed in more detail in the [deployment guide for Windows](#).

[< Running qmake](#)

[qmake Language >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

#### Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

#### Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

#### Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

#### Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

#### About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company





# Qt Documentation

Google Search Documentation



Qt 5.11 > qmake Manual > qmake Language

## Contents



### Operators

- [Assigning Values](#)
- [Appending Values](#)
- [Removing Values](#)
- [Adding Unique Values](#)
- [Replacing Values](#)
- [Variable Expansion](#)
- [Accessing qmake Properties](#)

### Scopes

- [Scope Syntax](#)
- [Scopes and Conditions](#)
- [Configuration and Scopes](#)
- [Platform Scope Values](#)

### Variables

- [Replace Functions](#)
- [Test Functions](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

### Reference



- [All Qt C++ Classes](#)
- [All QML Types](#)
- [All Qt Modules](#)
- [Qt Creator Manual](#)
- [All Qt Reference Documentation](#)

### Getting Started

- [Getting Started with Qt](#)
- [What's New in Qt 5](#)
- [Examples and Tutorials](#)
- [Supported Platforms](#)
- [Qt Licensing](#)

### Overviews

- [Development Tools](#)
- [User Interfaces](#)
- [Core Internals](#)
- [Data Storage](#)
- [Multimedia](#)
- [Networking and Connectivity](#)
- [Graphics](#)

# qmake Language

Many qmake project files simply describe the sources and header files used by the project, using a list of `name = value` and `name += value` definitions. qmake also provides other operators, functions, and scopes that can be used to process the information supplied in variable declarations. These advanced features allow Makefiles to be generated for multiple platforms from a single project file.

## Operators

In many project files, the assignment (`=`) and append (`+=`) operators can be used to include all the information about a project. The typical pattern of use is to assign a list of values to a variable, and append more values depending on the result of various tests. Since qmake defines certain variables using default values, it is sometimes necessary to use the removal (`-=`) operator to filter out values that are not required. The following sections describe how to use operators to manipulate the contents of variables.

## Assigning Values

The `=` operator assigns a value to a variable:

```
TARGET = myapp
```



赋值

The above line sets the `TARGET` variable to `myapp`. This will overwrite any values previously set for `TARGET` with `myapp`.

## Appending Values

The += operator appends a new value to the list of values in a variable:

```
DEFINES += USE_MY_STUFF
```

追加

The above line appends USE\_MY\_STUFF to the list of pre-processor defines to be put in the generated Makefile.

## Removing Values

The -= operator removes a value from the list of values in a variable:

```
DEFINES -= USE_MY_STUFF
```

移除

The above line removes USE\_MY\_STUFF from the list of pre-processor defines to be put in the generated Makefile.

## Adding Unique Values

The \*= operator adds a value to the list of values in a variable, but only if it is not already present. This prevents values from being included many times in a variable. For example:

```
DEFINES *= USE_MY_STUFF
```

添加未赋值的变量

In the above line, USE\_MY\_STUFF will only be added to the list of pre-processor defines if it is not already defined. Note that the `unique()` function can also be used to ensure that a variable only contains one instance of each value.

## Replacing Values

The `~=` operator replaces any values that match a regular expression with the specified value:

```
DEFINES ~= s/QT_[DT].+/QT
```

移除符合正则表达式的变量

In the above line, any values in the list that start with `QT_D` or `QT_T` are replaced with `QT`.

## Variable Expansion

The `$$` operator is used to extract the contents of a variable, and can be used to pass values between variables or supply them to functions:

```
EVERYTHING = $$SOURCES $$HEADERS  
message("The project contains the following files:")  
message($$EVERYTHING)
```

变量的展开

Variables can be used to store the contents of environment variables. These can be evaluated at the time when `qmake` is run, or included in the generated Makefile for evaluation when the project is built.

To obtain the contents of an environment value when `qmake` is run, use the `$(...)` operator:

```
DESTDIR = $(PWD)  
message(The project will be installed in $DESTDIR)
```

获取环境变量

In the above assignment, the value of the `PWD` environment variable is read when the project file is processed.

To obtain the contents of an environment value at the time when the generated Makefile is processed, use the `$(...)` operator:

```
DESTDIR = $$ (PWD)
message(The project will be installed in $$DESTDIR)
```

第一种为立即读取

```
DESTDIR = $(PWD)
message(The project will be installed in the value of PWD)
message(when the Makefile is processed.)
```

在Makefile中才进行展开

In the above assignment, the value of `PWD` is read immediately when the project file is processed, but `$(PWD)` is assigned to `DESTDIR` in the generated Makefile. This makes the build process more flexible as long as the environment variable is set correctly when the Makefile is processed.

## Accessing qmake Properties

The special `$$[...]` operator can be used to access qmake properties:

```
message(Qt version: $$[QT_VERSION])
message(Qt is installed in $$[QT_INSTALL_PREFIX])
message(Qt resources can be found in the following locations:)
message(Documentation: $$[QT_INSTALL_DOCS])
message(Header files: $$[QT_INSTALL_HEADERS])
message(Libraries: $$[QT_INSTALL_LIBS])
message(Binary files (executables): $$[QT_INSTALL_BINS])
message(Plugins: $$[QT_INSTALL_PLUGINS])
message(Data files: $$[QT_INSTALL_DATA])
message(Translation files: $$[QT_INSTALL_TRANSLATIONS])
message(Settings: $$[QT_INSTALL_CONFIGURATION])
message(Examples: $$[QT_INSTALL_EXAMPLES])
```

qmake预定义的属性

For more information, see [Configuring qmake](#).

The properties accessible with this operator are typically used to enable third party plugins and components to be integrated in Qt. For example, a *Qt Designer* plugin can be installed alongside *Qt Designer's* built-in plugins if the following declaration is made in its project file:

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

← 主要就是为了插件的方式

## Scopes

Scopes are similar to `if` statements in procedural programming languages. If a certain condition is true, the declarations inside the scope are processed.

### Scope Syntax

Scopes consist of a condition followed by an opening brace on the same line, a sequence of commands and definitions, and a closing brace on a new line:

```
<condition> {  
    <command or definition>  
    ...  
}
```

← 条件检测，注意第一个大括号必须在同一行

The opening brace *must be written on the same line as the condition*. Scopes may be concatenated to include more than one condition, as described in the following sections.

### Scopes and Conditions

A scope is written as a condition followed by a series of declarations contained within a pair of braces. For example:

```
win32 {  
    SOURCES += paintwidget_win.cpp  
}
```

The above code will add the `paintwidget_win.cpp` file to the sources listed in the generated Makefile when building for a Windows platform. When building for other platforms, the define will be ignored.

The conditions used in a given scope can also be negated to provide an alternative set of declarations that will be processed only if the original condition is false. For example, to process something when building for all platforms *except* Windows, negate the scope like this:

```
!win32 {  
    SOURCES -= paintwidget_win.cpp  
}
```

Scopes can be nested to combine more than one condition. For instance, to include a particular file for a certain platform only if debugging is enabled, write the following:

```
macx {  
    CONFIG(debug, debug|release) {  
        HEADERS += debugging.h  
    }  
}
```

多条件的连接



To save writing many nested scopes, you can nest scopes using the `:` operator. The nested scopes in the above example can be rewritten in the following way:



following way.

```
macx:CONFIG(debug, debug|release) {  
    HEADERS += debugging.h  
}
```

You may also use the `:` operator to perform single line conditional assignments. For example:

```
win32:DEFINES += USE_MY_STUFF
```

The above line adds `USE_MY_STUFF` to the `DEFINES` variable only when building for the Windows platform. Generally, the `:` operator behaves like a logical AND operator, joining together a number of conditions, and requiring all of them to be true.

There is also the `|` operator to act like a logical OR operator, joining together a number of conditions, and requiring only one of them to be true.

```
win32|macx {  
    HEADERS += debugging.h  
}
```

逻辑或的功能



You can also provide alternative declarations to those within a scope by using an `else` scope. Each `else` scope is processed if the conditions for the preceding scopes are false. This allows you to write complex tests when combined with other scopes (separated by the `:` operator as above). For example:

```
win32:xml {  
    message(Building for Windows)  
    SOURCES += xmlhandler_win.cpp  
} else:xml {
```

else语句的使用



```
SOURCES += xmlhandler.cpp
} else {
    message("Unknown configuration")
}
```

## Configuration and Scopes

The values stored in the **CONFIG** variable are treated specially by qmake. Each of the possible values can be used as the condition for a scope. For example, the list of values held by CONFIG can be extended with the `opengl` value:

```
CONFIG += opengl
```

As a result of this operation, any scopes that test for `opengl` will be processed. We can use this feature to give the final executable an appropriate name:

```
opengl {
    TARGET = application-gl
} else {
    TARGET = application
}
```

This feature makes it easy to change the configuration for a project without losing all the custom settings that might be needed for a specific configuration. In the above code, the declarations in the first scope are processed, and the final executable will be called `application-gl`. However, if `opengl` is not specified, the declarations in the second scope are processed instead, and the final executable will be called `application`.

Since it is possible to put your own values on the CONFIG line, this provides you with a convenient way to customize project files and fine-tune the generated Makefiles.

## Platform Scope Values

In addition to the `win32`, `macx`, and `unix` values used in many scope conditions, various other built-in platform and compiler-specific values can be tested with scopes. These are based on platform specifications provided in Qt's `mkspecs` directory. For example, the following lines from a project file show the current specification in use and test for the `linux-g++` specification:

```
message($$QMAKESPEC)

linux-g++ {
    message(Linux)
}
```

平台和编译器选项

You can test for any other platform-compiler combination as long as a specification exists for it in the `mkspecs` directory.

## Variables

Many of the variables used in project files are special variables that `qmake` uses when generating Makefiles, such as `DEFINES`, `SOURCES`, and `HEADERS`. In addition, you can create variables for your own use. `qmake` creates new variables with a given name when it encounters an assignment to that name. For example:

```
MY_VARIABLE = value
```

自由变量的创建

There are no restrictions on what you do to your own variables, as `qmake` will ignore them unless it needs to evaluate them when processing a scope.

You can also assign the value of a current variable to another variable by prefixing `$$` to the variable name. For example:

```
MY_DEFINES = $$DEFINES
```

Now the MY\_DEFINES variable contains what is in the DEFINES variable at this point in the project file. This is also equivalent to:

```
MY_DEFINES = ${DEFINES}
```

← 变量值的引用和展开

The second notation allows you to append the contents of the variable to another value without separating the two with a space. For example, the following will ensure that the final executable will be given a name that includes the project template being used:

```
TARGET = myproject_${TEMPLATE}
```

## Replace Functions

qmake provides a selection of built-in functions to allow the contents of variables to be processed. These functions process the arguments supplied to them and return a value, or list of values, as a result. To assign a result to a variable, use the \$\$ operator with this type of function as you would to assign contents of one variable to another:

```
HEADERS = model.h  
HEADERS += $$OTHER_HEADERS  
HEADERS = $$unique(HEADERS)
```

← 使用双\$\$符号，对函数的结果进行引用。

This type of function should be used on the right-hand side of assignments (that is, as an operand).

You can define your own functions for processing the contents of variables as follows:

```
defineReplace(functionName){  
    #function code  
}
```

The following example function takes a variable name as its only argument, extracts a list of values from the variable with the `eval()` built-in function, and compiles a list of files:

```
defineReplace(headersAndSources) {  
    variable = $$1  
    names = $$eval($$variable)  
    headers =  
    sources =  
  
    for(name, names) {  
        header = ${name}.h  
        exists($$header) {  
            headers += $$header  
        }  
        source = ${name}.cpp  
        exists($$source) {  
            sources += $$source  
        }  
    }  
    return($$headers $$sources)  
}
```

自定义函数的方式。

## Test Functions

qmake provides built-in functions that can be used as conditions when writing scopes. These functions do not return a value, but instead indicate

*success or failure:*

```
count(options, 2) {  
    message(Both release and debug specified.)  
}
```

This type of function should be used in conditional expressions only.

It is possible to define your own functions to provide conditions for scopes. The following example tests whether each file in a list exists and returns true if they all exist, or false if not:

```
defineTest(allFiles) {  
    files = $$ARGS  
  
    for(file, files) {  
        !exists($$file) {  
            return(false)  
        }  
    }  
    return(true)  
}
```

定义自己的条件检测函数

< Platform Notes

Advanced Usage >

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Advanced Usage](#)

## Contents



[Adding New Configuration Features](#)

[Installing Files](#)

[Adding Custom Targets](#)

[Adding Compilers](#)

[Library Dependencies](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)

[Qt Creator Manual](#)

[All Qt Reference Documentation](#)



## Getting Started

[Getting Started with Qt](#)

[What's New in Qt 5](#)

[Examples and Tutorials](#)

[Supported Platforms](#)

[Qt Licensing](#)

## Overviews

[Development Tools](#)

[User Interfaces](#)

[Core Internals](#)

[Data Storage](#)

[Multimedia](#)

[Networking and Connectivity](#)

[Graphics](#)

[Mobile APIs](#)

[QML Applications](#)

[All Qt Overviews](#)

# Advanced Usage

## Adding New Configuration Features

qmake lets you create your own features that can be included in project files by adding their names to the list of values specified by the `CONFIG` variable. Features are collections of custom functions and definitions in `.prf` files that can reside in one of many standard directories. The locations of these directories are defined in a number of places, and qmake checks each of them in the following order when it looks for `.prf` files:

1. In a directory listed in the `QMAKEFEATURES` environment variable that contains a list of directories delimited by the platform's path list separator (colon for Unix, semicolon for Windows).
2. In a directory listed in the `QMAKEFEATURES` property variable that contains a list of directories delimited by the platform's path list separator.
3. In a features directory residing within a `mkspecs` directory. `mkspecs` directories can be located beneath any of the directories listed in the `QMAKEPATH` environment variable that contains a list of directories delimited by the platform's path list separator. For example: `$QMAKEPATH/mkspecs/<features>`.
4. In a features directory residing beneath the directory provided by the `QMAKESPEC` environment variable. For example: `$QMAKESPEC/<features>`.
5. In a features directory residing in the `data_install/mkspecs` directory. For example: `data_install/mkspecs/<features>`.
6. In a features directory that exists as a sibling of the directory specified by the `QMAKESPEC` environment variable. For example: `$QMAKESPEC/../../<features>`.

The following features directories are searched for features files:

1. `features/unix`, `features/win32`, or `features/macx`, depending on the platform in use
2. `features/`

For example, consider the following assignment in a project file:

```
CONFIG += myfeatures
```

With this addition to the `CONFIG` variable, qmake will search the locations listed above for the `myfeatures.prf` file after it has finished parsing your project file. On Unix systems, it will look for the following file:

1. `$QMAKEFEATURES/myfeatures.prf` (for each directory listed in the `QMAKEFEATURES` environment variable)

2. `$$QMAKEFEATURES/myfeatures.prf` (for each directory listed in the `QMAKEFEATURES` property variable)
3. `myfeatures.prf` (in the project's root directory)
4. `$QMAKEPATH/mkspecs/features/unix/myfeatures.prf` and `$QMAKEPATH/mkspecs/features/myfeatures.prf` (for each directory listed in the `QMAKEPATH` environment variable)
5. `$QMAKESPEC/features/unix/myfeatures.prf` and `$QMAKESPEC/features/myfeatures.prf`
6. `data_install/mkspecs/features/unix/myfeatures.prf` and `data_install/mkspecs/features/myfeatures.prf`
7. `$QMAKESPEC/../../features/unix/myfeatures.prf` and `$QMAKESPEC/../../features/myfeatures.prf`

**Note:** The `.prf` files must have names in lower case.

## Installing Files

It is common on Unix to also use the build tool to install applications and libraries; for example, by invoking `make install`. For this reason, `qmake` has the concept of an `install` set, an object which contains instructions about the way a part of a project is to be installed. For example, a collection of documentation files can be described in the following way:

```
documentation.path = /usr/local/program/doc
documentation.files = docs/*
```

文档的安装，`path`指定安装的路径，`files`指定安装的内容。

The `path` member informs `qmake` that the files should be installed in `/usr/local/program/doc` (the `path` member), and the `files` member specifies the files that should be copied to the installation directory. In this case, everything in the `docs` directory will be copied to `/usr/local/program/doc`.

Once an install set has been fully described, you can append it to the install list with a line like this:

```
INSTALLS += documentation
```

最后在`INSTALLS`变量中追加文件

qmake will ensure that the specified files are copied to the installation directory. If you require more control over this process, you can also provide a definition for the extra member of the object. For example, the following line tells qmake to execute a series of commands for this install set:

```
unix:documentation.extra = create_docs; mv master.doc toc.doc
```

更加精细的控制

The `unix` scope ensures that these particular commands are only executed on Unix platforms. Appropriate commands for other platforms can be defined using other scope rules.

Commands specified in the extra member are executed before the instructions in the other members of the object are performed.

If you append a built-in install set to the `INSTALLS` variable and do not specify files or extra members, qmake will decide what needs to be copied for you. Currently, the `target` and `dlltarget` install sets are supported. For example:

```
target.path = /usr/local/myprogram  
INSTALLS += target
```

当前target和dlltarget安装是支持的。

In the above lines, qmake knows what needs to be copied, and will handle the installation process automatically.

## Adding Custom Targets

qmake tries to do everything expected of a cross-platform build tool. This is often less than ideal when you really need to run special platform-dependent commands. This can be achieved with specific instructions to the different qmake backends.

Customization of the Makefile output is performed through an object-style API as found in other places in qmake. Objects are defined automatically by specifying their *members*. For example:

```
mytarget.target = .buildfile
mytarget.commands = touch $$mytarget.target
mytarget.depends = mytarget2

mytarget2.commands = @echo Building $$mytarget.target
```

The definitions above define a qmake target called `mytarget`, containing a Makefile target called `.buildfile` which in turn is generated with the `touch()` function. Finally, the `.depends` member specifies that `mytarget` depends on `mytarget2`, another target that is defined afterwards. `mytarget2` is a dummy target. It is only defined to echo some text to the console.

The final step is to use the `QMAKE_EXTRA_TARGETS` variable to instruct qmake that this object is a target to be built:

```
QMAKE_EXTRA_TARGETS += mytarget mytarget2
```

← 指定创建多个目标对象

This is all you need to do to actually build custom targets. Of course, you may want to tie one of these targets to the `qmake build target`. To do this, you simply need to include your Makefile target in the list of `PRE_TARGETDEPS`.

Custom target specifications support the following members:

Member	Description
commands	The commands for generating the custom build target.
CONFIG	Specific configuration options for the custom build target. Can be set to <code>recursive</code> to indicate that rules should be created in the Makefile to call the relevant target inside the sub-target specific Makefile. This member defaults to creating an entry for each of the sub-targets.
depends	The existing build targets that the custom build target depends on.
recurse	Specifies which sub-targets should be used when creating the rules in the Makefile to call in the sub-target specific Makefile. This member is used only when <code>recursive</code> is set in <code>CONFIG</code> . Typical values are "Debug" and "Release".
recurse_target	Specifies the target that should be built via the sub-target Makefile for the rule in the Makefile. This member adds something like <code>\$(MAKE) -f Makefile [subtarget] [recurse_target]</code> . This member is used only when <code>recursive</code> is set in

	<code>\$(MAKE) -f Makefile.[subtarget] [recursive_target]</code> . This member is used only when <code>RECURSIVE</code> is set in <code>CONFIG</code> .
target	The name of the custom build target.

## Adding Compilers

添加新的编译器

It is possible to customize qmake to support new compilers and preprocessors:

```
new_moc.output = moc_{$QMAKE_FILE_BASE}.cpp
new_moc.commands = moc {$QMAKE_FILE_NAME} -o {$QMAKE_FILE_OUT}
new_moc.depend_command = g++ -E -M {$QMAKE_FILE_NAME} | sed "s,^.*: ,,"
new_moc.input = NEW_HEADERS
QMAKE_EXTRA_COMPILERS += new_moc
```

With the above definitions, you can use a drop-in replacement for `moc` if one is available. The command is executed on all arguments given to the `NEW_HEADERS` variable (from the `input` member), and the result is written to the file defined by the `output` member. This file is added to the other source files in the project. Additionally, `qmake` will execute `depend_command` to generate dependency information, and place this information in the project as well.

Custom compiler specifications support the following members:

Member	Description
commands	The commands used for for generating the output from the input.
CONFIG	Specific configuration options for the custom compiler. See the <code>CONFIG</code> table for details.
depend_command	Specifies a command used to generate the list of dependencies for the output.
dependency_type	Specifies the type of file the output is. If it is a known type (such as <code>TYPE_C</code> , <code>TYPE_UI</code> , <code>TYPE_QRC</code> ), it is handled as one of those type of files.

depends	Specifies the dependencies of the output file.
input	The variable that specifies the files that should be processed with the custom compiler.
name	A description of what the custom compiler is doing. This is only used in some backends.
output	The filename that is created from the custom compiler.
output_function	Specifies a custom qmake function that is used to specify the filename to be created.
variables	Indicates that the variables specified here are replaced with <code>\$(QMAKE_COMP_VARNAME)</code> when referred to in the pro file as <code>\$(VARNAME)</code> .
variable_out	The variable that the files created from the output should be added to.

The CONFIG member supports the following options:

Option	Description
combine	Indicates that all of the input files are combined into a single output file.
target_predeps	Indicates that the output should be added to the list of <code>PRE_TARGETDEPS</code> .
explicit_dependencies	The dependencies for the output only get generated from the depends member and from nowhere else.
no_link	Indicates that the output should not be added to the list of objects to be linked in.

## Library Dependencies

Often when linking against a library, qmake relies on the underlying platform to know what other libraries this library links against, and lets the platform pull them in. In many cases, however, this is not sufficient. For example, when statically linking a library, no other libraries are linked to, and therefore no dependencies to those libraries are created. However, an application that later links against this library will need to know where to find the symbols that the static library will require. qmake attempts to keep track of the dependencies of a library, where appropriate, if you explicitly enable tracking.

The first step is to enable dependency tracking in the library itself. To do this you must tell qmake to save information about the library:

```
CONFIG += create_prl
```

This is only relevant to the `lib` template, and will be ignored for all others. When this option is enabled, qmake will create a file ending in `.prl` which will save some meta-information about the library. This metafile is just like an ordinary project file, but only contains internal variable declarations. When installing this library, by specifying it as a target in an `INSTALLS` declaration, qmake will automatically copy the `.prl` file to the installation path.

The second step in this process is to enable reading of this meta information in the applications that use the static library:

```
CONFIG += link_prl
```

When this is enabled, qmake will process all libraries linked to by the application and find their meta-information. qmake will use this to determine the relevant linking information, specifically adding values to the application project file's list of `DEFINES` as well as `LIBS`. Once qmake has processed this file, it will then look through the newly introduced libraries in the `LIBS` variable, and find their dependent `.prl` files, continuing until all libraries have been resolved. At this point, the Makefile is created as usual, and the libraries are linked explicitly against the application.

The `.prl` files should be created by qmake only, and should not be transferred between operating systems, as they may contain platform-dependent information.

[< qmake Language](#)

[Using Precompiled Headers >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

[Download](#)

[Product](#)

[Services](#)

[Developers](#)

[About us](#)



[Start for Free](#)

[Qt for Application Development](#)

[Qt for Device Creation](#)

[Qt Open Source](#)

[Terms & Conditions](#)

[Licensing FAQ](#)

[Qt in Use](#)

[Qt for Application Development](#)

[Qt for Device Creation](#)

[Commercial Features](#)

[Qt Creator IDE](#)

[Qt Quick](#)

[Technology Evaluation](#)

[Proof of Concept](#)

[Design & Implementation](#)

[Productization](#)

[Qt Training](#)

[Partner Network](#)

[Documentation](#)

[Examples & Tutorials](#)

[Development Tools](#)

[Wiki](#)

[Forums](#)

[Contribute to Qt](#)

[Training & Events](#)

[Resource Center](#)

[News](#)

[Careers](#)

[Locations](#)

[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > Using Precompiled Headers

## Contents



[Adding Precompiled Headers to Your Project](#)

[Project Options](#)

[Notes on Possible Issues](#)

[Example Project](#)

[mydialog.ui](#)

[stable.h](#)

[myobject.h](#)

[myobject.cpp](#)

[util.cpp](#)

[main.cpp](#)

[precompile.pro](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)  
[All QML Types](#)  
[All Qt Modules](#)  
[Qt Creator Manual](#)  
[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)  
[What's New in Qt 5](#)  
[Examples and Tutorials](#)  
[Supported Platforms](#)  
[Qt Licensing](#)

## Overviews

[Development Tools](#)  
[User Interfaces](#)  
[Core Internals](#)  
[Data Storage](#)  
[Multimedia](#)  
[Networking and Connectivity](#)  
[Graphics](#)  
[Mobile APIs](#)  
[QML Applications](#)  
[All Qt Overviews](#)

# Using Precompiled Headers

Precompiled headers (PCH) are a performance feature supported by some compilers to compile a stable body of code, and store the compiled state of the code in a binary file. During subsequent compilations, the compiler will load the stored state, and continue compiling the specified file. Each subsequent compilation is faster because the stable code does not need to be recompiled.

qmake supports the use of precompiled headers on some platforms and build environments, including:

- › Windows
  - › nmake
  - › Visual Studio projects (VS 2008 and later)
- › macOS, iOS, tvOS, and watchOS
  - › Makefile
  - › Xcode
- › Unix
  - › GCC 3.4 and above

## Adding Precompiled Headers to Your Project

添加预编译的头文件

The precompiled header must contain code which is *stable* and *static* throughout your project. A typical precompiled header might look like this:

```
// Add C includes here

#if defined __cplusplus
// Add C++ includes here
#include <stdlib>
#include <iostream>
```

```
#include <vector>
#include <QApplication> // Qt includes
#include <QPushButton>
#include <QLabel>
#include "thirdparty/include/libmain.h"
#include "my_stable_class.h"
...
#endif
```

**Note:** A precompiled header file needs to separate C includes from C++ includes, since the precompiled header file for C files may not contain C++ code.

## Project Options

To make your project use precompiled headers, you only need to define the `PRECOMPILED_HEADER` variable in your project file:

```
PRECOMPILED_HEADER = stable.h
```

使用预编译头文件的方式

qmake will handle the rest, to ensure the creation and use of the precompiled header file. You do not need to include the precompiled header file in `HEADERS`, as qmake will do this if the configuration supports precompiled headers.

The MSVC and g++ specs targeting Windows enable `precompile_header` by default.

Using this option, you may trigger conditional blocks in your project file to add settings when using precompiled headers. For example:

```
precompile_header:!isEmpty(PRECOMPILED_HEADER) {
  DEFINES += USING_PCH
}
```

To use the precompiled header also for C files on MSVC nmake target, add `precompile_header_c` to the `CONFIG` variable. If the header is used also for C++ and it contains C++ keywords/includes, enclose them with `#ifdef __cplusplus`).

## Notes on Possible Issues

On some platforms, the file name suffix for precompiled header files is the same as that for other object files. For example, the following declarations may cause two different object files with the same name to be generated:

```
PRECOMPILED_HEADER = window.h
SOURCES             = window.cpp
```

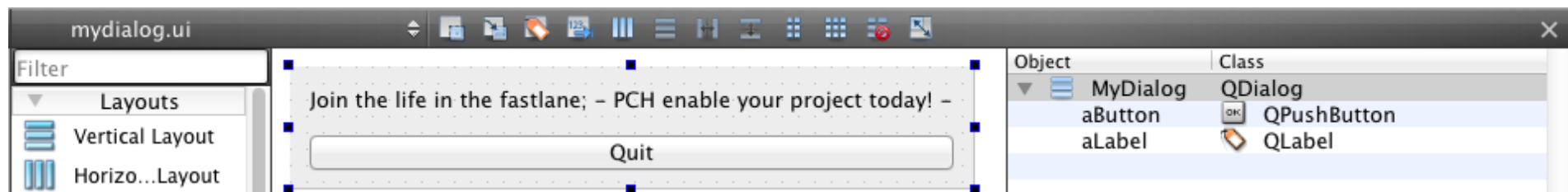
To avoid potential conflicts like these, give distinctive names to header files that will be precompiled.

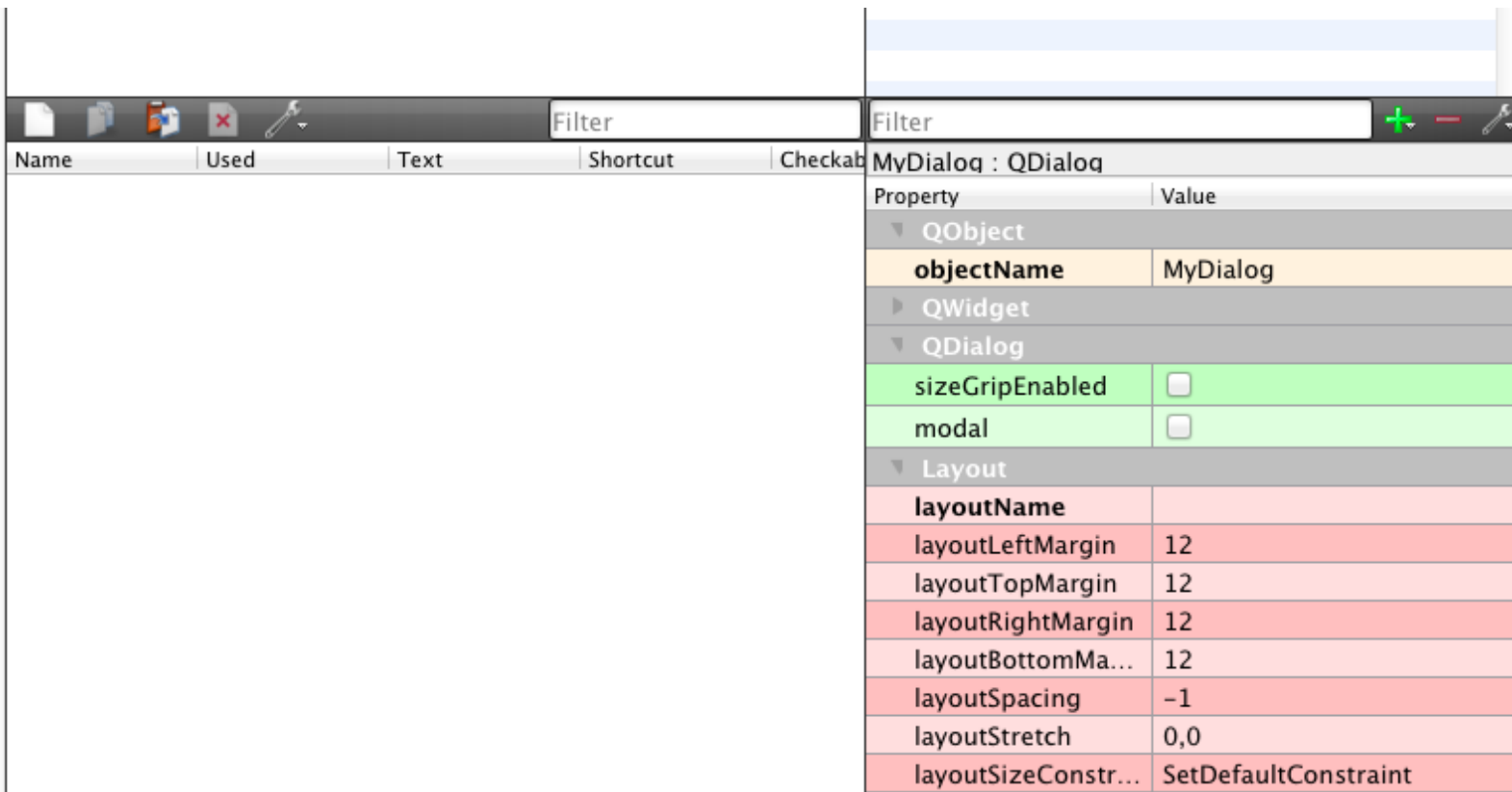
## Example Project

You can find the following source code in the `examples/qmake/precompile` directory in the Qt distribution:

### mydialog.ui

The following image displays the `mydialog.ui` file in Qt Creator Design mode. You can view the code in the Edit mode.





stable.h

```
/* Add C includes here */

#if defined __cplusplus
/* Add C++ includes here */

# include <iostream>
# include <QApplication>
# include <QPushButton>
# include <QLabel>
```

```
#endif
```

## myobject.h

```
#include <QObject>

class MyObject : public QObject
{
public:
    MyObject();
    ~MyObject();
};
```

## myobject.cpp

```
#include <iostream>
#include <QDebug>
#include <QObject>
#include "myobject.h"

MyObject::MyObject()
    : QObject()
{
    std::cout << "MyObject::MyObject()\n";
}
```

## util.cpp



```
void util_function_does_nothing()
{
    // Nothing here...
    int x = 0;
    ++x;
}
```

## main.cpp

```
#include <QApplication>
#include <QPushButton>
#include <QLabel>
#include "myobject.h"
#include "mydialog.h"

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    MyObject obj;
    MyDialog dialog;

    dialog.connect(dialog.aButton, SIGNAL(clicked()), SLOT(close()));
    dialog.show();

    return app.exec();
}
```

precompile pro

```
TEMPLATE = app
LANGUAGE = C++
CONFIG += console precompile_header
CONFIG -= app_bundle
```

```
# Use Precompiled headers (PCH)
PRECOMPILED_HEADER = stable.h
```

预编译头文件

```
HEADERS = stable.h \
          mydialog.h \
          myobject.h
SOURCES = main.cpp \
          mydialog.cpp \
          myobject.cpp \
          util.cpp
FORMS = mydialog.ui
```

[< Advanced Usage](#)

[Configuring qmake >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

**Download**

[Start for Free](#)

**Product**

[Qt in Use](#)

**Services**

[Technology Evaluation](#)

**Developers**

[Documentation](#)

**About us**

[Training & Events](#)

[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Configuring qmake](#)

## Contents



[Properties](#)

[QMAKESPEC](#)

[Cache File](#)

[File Extensions](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)

[Qt Creator Manual](#)

[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)

[What's New in Qt 5](#)

[Examples and Tutorials](#)

[Supported Platforms](#)

[Qt Licensing](#)

Overviews

[Development Tools](#)

[User Interfaces](#)

[Core Internals](#)

[Data Storage](#)

[Multimedia](#)

[Networking and Connectivity](#)

[Graphics](#)

[Mobile APIs](#)

[QML Applications](#)

[All Qt Overviews](#)

# Configuring qmake

## Properties

qmake has a system for persistent configuration, which allows you to set a property in qmake once, and query it each time qmake is invoked. You can set a property in qmake as follows:

set a property in qmake as follows:

```
qmake -set PROPERTY VALUE
```

The appropriate property and value should be substituted for PROPERTY and VALUE.

You can retrieve this information back from qmake as follows:

```
qmake -query PROPERTY  
qmake -query #queries all current PROPERTY/VALUE pairs
```

**Note:** `qmake -query` lists built-in properties in addition to the properties that you set with `qmake -set PROPERTY VALUE`.

This information will be saved into a [QSettings](#) object (meaning it will be stored in different places for different platforms).

The following list summarizes the `built-in` properties:

- › `QMAKE_SPEC` - the shortname of the host `mkspec` that is resolved and stored in the `QMAKESPEC` variable during a host build
- › `QMAKE_VERSION` - the current version of qmake
- › `QMAKE_XSPEC` - the shortname of the target `mkspec` that is resolved and stored in the `QMAKESPEC` variable during a target build
- › `QT_HOST_BINS` - location of host executables
- › `QT_HOST_DATA` - location of data for host executables used by qmake
- › `QT_HOST_PREFIX` - default prefix for all host paths
- › `QT_INSTALL_ARCHDATA` - location of general architecture-dependent Qt data
- › `QT_INSTALL_BINS` - location of Qt binaries (tools and applications)
- › `QT_INSTALL_CONFIGURATION` - location for Qt settings. Not applicable on Windows
- › `QT_INSTALL_DATA` - location of general architecture-independent Qt data
- › `QT_INSTALL_DOCS` - location of documentation

- › `QT_INSTALL_DOCS` - location of documentation
- › `QT_INSTALL_EXAMPLES` - location of examples
- › `QT_INSTALL_HEADERS` - location for all header files
- › `QT_INSTALL_IMPORTS` - location of QML 1.x extensions
- › `QT_INSTALL_LIBEXECS` - location of executables required by libraries at runtime
- › `QT_INSTALL_LIBS` - location of libraries
- › `QT_INSTALL_PLUGINS` - location of Qt plugins
- › `QT_INSTALL_PREFIX` - default prefix for all paths
- › `QT_INSTALL_QML` - location of QML 2.x extensions
- › `QT_INSTALL_TESTS` - location of Qt test cases
- › `QT_INSTALL_TRANSLATIONS` - location of translation information for Qt strings
- › `QT_SYSDIR` - the sysroot used by the target build environment
- › `QT_VERSION` - the Qt version. We recommend that you query Qt module specific version numbers by using `$$QT.<module>.version` variables instead.

For example, you can query the installation of Qt for this version of qmake with the `QT_INSTALL_PREFIX` property:

```
qmake -query "QT_INSTALL_PREFIX"
```

You can query the values of properties in a project file as follows:

```
QMAKE_VERS = $$[QMAKE_VERSION]
```

## QMAKESPEC

qmake requires a platform and compiler description file which contains many default values used to generate appropriate Makefiles. The standard Qt distribution comes with many of these files, located in the `mkspecs` subdirectory of the Qt installation.

The `QMAKESPEC` environment variable can contain any of the following:

- › A complete path to a directory containing a `qmake.conf` file. In this case qmake will open the `qmake.conf` file from within that directory. If the file does not exist, qmake will exit with an error.
- › The name of a platform-compiler combination. In this case, qmake will search in the directory specified by the `mkspecs` subdirectory of the data path specified when Qt was compiled (see [QLibraryInfo::DataPath](#)).

**Note:** The `QMAKESPEC` path will be automatically added to the generated Makefile after the contents of the `INCLUDEPATH` system variable.

## Cache File

The cache file is a special file qmake reads to find settings not specified in the `qmake.conf` file, project files, or at the command line. When qmake is run, it looks for a file called `.qmake.cache` in parent directories of the current directory, unless you specify `-nocache`. If qmake fails to find this file, it will silently ignore this step of processing.

If qmake finds a `.qmake.cache` file then it will process this file first before it processes the project file.

## File Extensions

Under normal circumstances qmake will try to use appropriate file extensions for your platform. However, it is sometimes necessary to override the default choices for each platform and explicitly define file extensions for qmake to use. This is achieved by redefining certain built-in variables. For example, the extension used for `.moc` files can be redefined with the following assignment in a project file:

```
QMAKE_EXT_MOC = .mymoc
```

The following variables can be used to redefine common file extensions recognized by qmake:



- › `QMAKE_EXT_MOC` modifies the extension placed on included moc files.
- › `QMAKE_EXT_UI` modifies the extension used for *Qt Designer* UI files (usually in `FORMS`).
- › `QMAKE_EXT_PRL` modifies the extension placed on library dependency files.
- › `QMAKE_EXT_LEX` changes the suffix used in Lex files (usually in `LEXSOURCES`).
- › `QMAKE_EXT_YACC` changes the suffix used in Yacc files (usually in `YACCSOURCES`).
- › `QMAKE_EXT_OBJ` changes the suffix used on generated object files.

All of the above accept just the first value, so you must assign to it just one value that will be used throughout your project file. There are two variables that accept a list of values:

- › `QMAKE_EXT_CPP` causes qmake to interpret all files with these suffixes as C++ source files.
- › `QMAKE_EXT_H` causes qmake to interpret all files with these suffixes as C and C++ header files.

[< Using Precompiled Headers](#)

[Reference >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

#### Download

Start for Free  
Qt for Application Development  
Qt for Device Creation  
Qt Open Source  
Terms & Conditions

#### Product

Qt in Use  
Qt for Application Development  
Qt for Device Creation  
Commercial Features  
Qt Creator IDE

#### Services

Technology Evaluation  
Proof of Concept  
Design & Implementation  
Productization  
Qt Training

#### Developers

Documentation  
Examples & Tutorials  
Development Tools  
Wiki  
Forums

#### About us

Training & Events  
Resource Center  
News  
Careers  
Locations

[Licensing FAQ](#)

[Qt Quick](#)

[Partner Network](#)

[Contribute to Qt](#)

[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Reference](#)

## Contents



[Variable Reference](#)

[Function Reference](#)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)

[All QML Types](#)

[All Qt Modules](#)

[Qt Creator Manual](#)

[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)

[What's New in Qt 5](#)

Examples and Tutorials

Supported Platforms

Qt Licensing

Overviews

Development Tools

User Interfaces

Core Internals

Data Storage

Multimedia

Networking and Connectivity

Graphics

Mobile APIs

QML Applications

All Qt Overviews

# Reference

The reference sections describe in detail the variables and functions that are available for use in qmake project files.

## Variable Reference

[Variables](#) describes the variables that are recognized by qmake when configuring the build process for projects.

## Function Reference

There are two types of qmake functions: replace functions and test functions. Replace functions return a value list, while test functions return a boolean result. The functions are implemented in two places: fundamental functionality is offered as built-in functions. More complex functions are implemented in a library of feature files (.prf).

The functions are divided into categories according to their type:

- › [Replace Functions](#)
- › [Test Functions](#)

replace函数和测试函数

[< Configuring qmake](#)

[Variables >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > Variables

## Contents



[CONFIG](#)  
[DEFINES](#)  
[DEF\\_FILE](#)  
[DEPENDPATH](#)  
[DESTDIR](#)  
[DISTFILES](#)  
[DLLDESTDIR](#)  
[FORMS](#)  
[GUID](#)  
[HEADERS](#)  
[ICON](#)  
[IDLSOURCES](#)  
[INCLUDEPATH](#)  
[INSTALLS](#)  
[LEXIMPLS](#)  
[LEXOBJECTS](#)

LEXSOURCES  
LIBS  
LITERAL\_HASH  
MAKEFILE  
MAKEFILE\_GENERATOR  
MSVCPROJ\_\*  
MOC\_DIR  
OBJECTIVE\_HEADERS  
OBJECTIVE\_SOURCES  
OBJECTS  
OBJECTS\_DIR  
POST\_TARGETDEPS  
PRE\_TARGETDEPS  
PRECOMPILED\_HEADER  
PWD  
OUT\_PWD  
QMAKE  
QMAKESPEC  
QMAKE\_AR\_CMD  
QMAKE\_BUNDLE\_DATA  
QMAKE\_BUNDLE\_EXTENSION  
QMAKE\_CC  
QMAKE\_CFLAGS  
QMAKE\_CFLAGS\_DEBUG



QMAKE\_CFLAGS\_RELEASE  
QMAKE\_CFLAGS\_SHLIB  
QMAKE\_CFLAGS\_THREAD  
QMAKE\_CFLAGS\_WARN\_OFF  
QMAKE\_CFLAGS\_WARN\_ON  
QMAKE\_CLEAN  
QMAKE\_CXX  
QMAKE\_CXXFLAGS  
QMAKE\_CXXFLAGS\_DEBUG  
QMAKE\_CXXFLAGS\_RELEASE  
QMAKE\_CXXFLAGS\_SHLIB  
QMAKE\_CXXFLAGS\_THREAD  
QMAKE\_CXXFLAGS\_WARN\_OFF  
QMAKE\_CXXFLAGS\_WARN\_ON  
QMAKE\_DEVELOPMENT\_TEAM  
QMAKE\_DISTCLEAN  
QMAKE\_EXTENSION\_SHLIB  
QMAKE\_EXTENSION\_STATICLIB  
QMAKE\_EXT\_MOC  
QMAKE\_EXT\_UI  
QMAKE\_EXT\_PRL  
QMAKE\_EXT\_LEX  
QMAKE\_EXT\_YACC  
QMAKE\_EXT\_OBJ

QMAKE\_EXT\_CPP  
QMAKE\_EXT\_H  
QMAKE\_EXTRA\_COMPILERS  
QMAKE\_EXTRA\_TARGETS  
QMAKE\_FAILED\_REQUIREMENTS  
QMAKE\_FRAMEWORK\_BUNDLE\_NAME  
QMAKE\_FRAMEWORK\_VERSION  
QMAKE\_HOST  
QMAKE\_INCDIR  
QMAKE\_INCDIR\_EGL  
QMAKE\_INCDIR\_OPENGL  
QMAKE\_INCDIR\_OPENGL\_ES2  
QMAKE\_INCDIR\_OPENVG  
QMAKE\_INCDIR\_X11  
QMAKE\_INFO\_PLIST  
QMAKE\_IOS\_DEPLOYMENT\_TARGET  
QMAKE\_LFLAGS  
QMAKE\_LFLAGS\_CONSOLE  
QMAKE\_LFLAGS\_DEBUG  
QMAKE\_LFLAGS\_PLUGIN  
QMAKE\_LFLAGS\_RPATH  
QMAKE\_LFLAGS\_REL\_RPATH  
QMAKE\_REL\_RPATH\_BASE  
QMAKE\_LFLAGS\_RPATHLINK

QMAKE\_LFLAGS\_RELEASE  
QMAKE\_LFLAGS\_APP  
QMAKE\_LFLAGS\_SHLIB  
QMAKE\_LFLAGS\_SONAME  
QMAKE\_LFLAGS\_THREAD  
QMAKE\_LFLAGS\_WINDOWS  
QMAKE\_LIBDIR  
QMAKE\_LIBDIR\_FLAGS  
QMAKE\_LIBDIR\_EGL  
QMAKE\_LIBDIR\_OPENGL  
QMAKE\_LIBDIR\_OPENVG  
QMAKE\_LIBDIR\_X11  
QMAKE\_LIBS  
QMAKE\_LIBS\_EGL  
QMAKE\_LIBS\_OPENGL  
QMAKE\_LIBS\_OPENGL\_ES1, QMAKE\_LIBS\_OPENGL\_ES2  
QMAKE\_LIBS\_OPENVG  
QMAKE\_LIBS\_THREAD  
QMAKE\_LIBS\_X11  
QMAKE\_LIB\_FLAG  
QMAKE\_LINK  
QMAKE\_LINK\_SHLIB\_CMD  
QMAKE\_LN\_SHLIB  
QMAKE\_OBJECTIVE\_CFLAGS

QMAKE\_POST\_LINK  
QMAKE\_PRE\_LINK  
QMAKE\_PROJECT\_NAME  
QMAKE\_PROVISIONING\_PROFILE  
QMAKE\_MAC\_SDK  
QMAKE\_MACOSX\_DEPLOYMENT\_TARGET  
QMAKE\_MAKEFILE  
QMAKE\_QMAKE  
QMAKE\_RESOURCE\_FLAGS  
QMAKE\_RPATHDIR  
QMAKE\_RPATHLINKDIR  
QMAKE\_RUN\_CC  
QMAKE\_RUN\_CC\_IMP  
QMAKE\_RUN\_CXX  
QMAKE\_RUN\_CXX\_IMP  
QMAKE\_SONAME\_PREFIX  
QMAKE\_TARGET  
QMAKE\_TARGET\_COMPANY  
QMAKE\_TARGET\_DESCRIPTION  
QMAKE\_TARGET\_COPYRIGHT  
QMAKE\_TARGET\_PRODUCT  
QMAKE\_TVOS\_DEPLOYMENT\_TARGET  
QMAKE\_UIC\_FLAGS  
QMAKE\_WATCHOS\_DEPLOYMENT\_TARGET

QT  
QTPLUGIN  
QT\_VERSION  
QT\_MAJOR\_VERSION  
QT\_MINOR\_VERSION  
QT\_PATCH\_VERSION  
RC\_FILE  
RC\_CODEPAGE  
RC\_DEFINES  
RC\_ICONS  
RC\_LANG  
RC\_INCLUDEPATH  
RCC\_DIR  
REQUIRES  
RESOURCES  
RES\_FILE  
SOURCES  
SUBDIRS  
TARGET  
TARGET\_EXT  
TARGET\_x  
TARGET\_x.y.z  
TEMPLATE  
TRANSLATIONS

UI\_DIR  
VERSION  
VERSION\_PE\_HEADER  
VER\_MAJ  
VER\_MIN  
VER\_PAT  
VPATH  
WINRT\_MANIFEST  
YACCSOURCES  
\_PRO\_FILE\_  
\_PRO\_FILE\_PWD\_

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

## Reference



[All Qt C++ Classes](#)  
[All QML Types](#)  
[All Qt Modules](#)  
[Qt Creator Manual](#)  
[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)

- What's New in Qt 5
- Examples and Tutorials
- Supported Platforms
- Qt Licensing

Overviews

- Development Tools
- User Interfaces
- Core Internals
- Data Storage
- Multimedia
- Networking and Connectivity
- Graphics
- Mobile APIs
- QML Applications
- All Qt Overviews

## 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, `QMAKE_LIBS` can be used to specify a list of libraries that a project needs to link against, and `QMAKE_LIBS_X11` can be

used to extend or override this list.

## CONFIG

指定项目的配置和编译。

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

The following CONFIG values control compilation 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 <i>both</i> 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.
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.

子目录顺序处理



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.
Option	<div> <div>C++11 support is enabled.</div> <div>This option has no effect if the compiler does not support C++11. By default, support is disabled.</div> </div>
c++14	<div> <div>C++14 support is enabled.</div> <div>This option has no effect if the compiler does not support C++14. By default, support is disabled.</div> </div>
depend_includepath	Appending the value of INCLUDEPATH to DEPENDPATH is enabled. Set by default.

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` and a `Makefile.Release`.

During the latter passes, `build_pass` and the respective `debug` or `release` option is appended to `CONFIG`. 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 <code>.prl</code> which will save meta-information about the library (see <a href="#">Library Dependencies</a> 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 <a href="#">Library Dependencies</a> for more info).

**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 <code>\1{#qt}{QT}</code> variable.
x11	The target is a X11 application or library. The proper include paths and libraries will automatically be added to the project.
testcase	The target is an automated test. A <a href="#">check target</a> will be added to the generated Makefile to run the test. Only relevant when generating Makefiles.
insignificant_test	The exit code of the automated test will be ignored. Only relevant if <code>testcase</code> 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.
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 <code>dll</code> can also be used on all platforms; a shared library file with the appropriate suffix for the target platform ( <code>.dll</code> or <code>.so</code> ) 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 <code>dll</code> as well.
designer	The target is a plugin for <i>Qt Designer</i> .
no_iflags_merge	Ensures that the list of libraries stored in the <code>LIBS</code> variable is not reduced to a list of unique values before it is used.

These options define specific features on Windows only:

Option	Description
flat	When using the <code>vcapp</code> 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.

The build process for bundles is also influenced by the contents of the [QMAKE\\_BUNDLE\\_DATA](#) variable.

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
```

## DEF\_FILE

**Note:** This variable is used only on Windows when using the app template.

**note:** This variable is used only on windows when using the app template.

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

## DEPENDPATH ← 头文件的依赖

Specifies a list of all directories to look in to resolve dependencies. This variable is used when crawling through included files.

## DESTDIR

Specifies where to put the target file. ← 目标对象的存储位置

For example:

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

## 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.

## FORMS

指定Ui 文件



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

头文件



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.

## IDL SOURCES

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

## INCLUDEPATH ← 指定头文件的搜索路径

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

指定安装的内容

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
```

For more information, see [Installing Files](#).

This variable is also used to specify which additional files will be deployed to embedded devices.

## 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



## 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

指定依赖的库

**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
```

By default, the list of libraries stored in `LIBS` is reduced to a list of unique names before it is used. To change this behavior, add the `no_lflags_merge` option to the `CONFIG` variable:

```
CONFIG += no_lflags_merge
```

## 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 ← 指定生成的Makefile的名字

**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

指定moc类文件的存放目录

Specifies the directory where all intermediate moc files should be placed.

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 ← 指定object中间文件的存放目录

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` ← 指定对象依赖的库

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` ← 预编译的头文件

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` ← 指定生成的Makefile的存放目录

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.

# 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 containing information about the headers supplied with the framework:

```
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.

## QMAKE\_BUNDLE\_EXTENSION

**Note:** This variable is used on [macOS](#), iOS, tvOS, and watchOS only.

Specifies the extension to be used for library bundles. This allows frameworks to be created with custom extensions instead of the standard `.framework` directory name extension.

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

```
QMAKE_BUNDLE_EXTENSION = .myframework
```

QMAKE\_CC ← 指定C编译器

[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 ← 指定C编译器的flags

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\_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 by qmake or [qmake.conf](#) and rarely needs to be modified.

### QMAKE\_CFLAGS\_WARN\_OFF



## 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 ← 指定C++编译器

[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 ← 指定C++编译器的编译flags

[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\_SHLIB

Specifies the C++ 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\_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



指定distclean的清空列表

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](#).

## QMAKE\_EXTRA\_TARGETS

Specifies a list of additional qmake targets.

See also [Adding Custom Targets](#).

## QMAKE\_FAILED\_REQUIREMENTS ← 包含失败的条件列表

**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 an [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

```
win32-g++:contains(QMAKE_HOST.arch, x86_64):{  
    message("Host is 64bit")  
    ...  
}
```

QMAKE\_INCDIR ← 指定头文件路径

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 a X11 target. The value of this variable is typically handled by qmake or [qmake.conf](#) and rarely needs to be modified.



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

**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

If building for iOS, and the `Info.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 documentation from Apple for more information regarding this key.

**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\_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

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

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\_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 typically handled by qmake or [qmake.conf](#) and rarely needs to be modified.

## 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 system library paths. 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 [qmake.conf](#) and rarely needs to be modified.

If the OpenGL implementation uses EGL (most OpenGL/ES systems), then `QMAKE_LIBDIR_EGL` may also need to 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 all project libraries. The value of this variable is typically handled by qmake or `qmake.conf` and rarely needs to be modified.

## 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

These variables specify all the OpenGL libraries for OpenGL ES 1 and OpenGL ES 2.

The value of these variables 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\_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.

## QMAKE\_LN\_SHLIB

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\_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.

**Note:** Specifying the provisioning profile disables the automatically managed signing.

## 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 Version Dependencies](#).

## 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

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 Linux and Darwin-based ones) and is detectable by checking whether [QMAKE\\_REL\\_RPATH\\_BASE](#) is set.

## 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
```

However, the prefix may be also specified using different placeholders, or an absolute path, such as one of the following:

```
# <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\_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. For example, `-no-stringliteral` can be passed to use [QLatin1String](#) instead of [QStringLiteral](#) in generated code (which is the default for dynamic libraries).

## 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](#).

## QT ← 指定所需要的模块

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](#).

## QTPLUGIN

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

Qt的版本信息

Contains the current version of Qt.

QT\_MAJOR\_VERSION

Qt的主版本信息

Contains the current major version of Qt.

QT\_MINOR\_VERSION

Qt的次版本信息

Contains the current minor version of Qt.

QT\_PATCH\_VERSION

Qt的这一版本的修订次数

Contains the current patch version of Qt.

RC\_FILE

Specifies the name of the resource file for the application. The value of this variable is typically handled by qmake or [qmake.conf](#) and rarely needs to be modified.

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
```

## RC\_ICONS

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

指定资源文件

**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

Specifies the name of the compiled Windows resource file for the target. The value of this variable is typically handled by qmake or [qmake.conf](#) and rarely needs to be modified.

## SOURCES

指定源文件列表

**Specifies the names of all source files in the project.**

For example:

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

See also [HEADERS](#).

## SUBDIRS ← 多个目录下处理方式

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 the order in which they are specified, update the `CONFIG` variable to include the `ordered` option:

```
CONFIG += ordered
```

It is possible to modify this default behavior of SUBDIRS by giving additional 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.
.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 ← 指定目标文件的名字

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

For example:

```
TEMPLATE = app
```

```
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

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

## TEMPLATE

指定目标文件的类型

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 <a href="#">Building an Application</a> for more information.

lib	Creates a Makefile for building libraries. See <a href="#">Building a Library</a> for more information.
subdirs	Creates a Makefile for building targets in subdirectories. The subdirectories are specified using the <a href="#">SUBDIRS</a> 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. <b>Note:</b> 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 <a href="#">Creating Visual Studio Project Files</a> 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 指定翻译文件

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

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

## UI\_DIR

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 指定对象的版本号

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\_PATCH

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

## VPATH

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](#).

## WINRT\_MANIFEST

Specifies parameters to be passed to the application manifest on [Windows Runtime](#). The allowed values are:

Member	Description
architecture	The target architecture. Defaults to <code>VCPP01_ARCH</code> .



architecture	The target architecture. Defaults to <code>VCXROS_ARCH</code> .
background	Tile background color. Defaults to <code>green</code> .
capabilities	Specifies capabilities to add to the capability list.
capabilities_device	Specifies device capabilities to add to the capability list (location, webcam, and so on).
CONFIG	Specifies additional flags for processing the input manifest file. Currently, <code>verbatim</code> is the only available option.
default_language	The default language code of the application. Defaults to <code>"en"</code> .
dependencies	Specifies dependencies required by the package.
description	Package description. Defaults to <code>Default package description</code> .
foreground	Tile foreground (text) color. Defaults to <code>light</code> . This option is only available for Windows Store apps on Windows 8 and Windows RT.
iconic_tile_icon	Image file for the <code>iconic</code> tile template icon. Default provided by the mkspec.
iconic_tile_small	Image file for the small <code>iconic</code> tile template logo. Default provided by the mkspec.
identity	The unique ID of the app. Defaults to reusing the existing generated manifest's UUID, or generates a new UUID if none is present.
logo_30x30	Logo image file of size 30x30 pixels. This is not supported on Windows Phone.
logo_41x41	Logo image file of size 41x41 pixels. This is only supported on Windows Phone.
logo_70x70	Logo image file of size 70x70 pixels. This is not supported on Windows Phone.
logo_71x71	Logo image file of size 71x71 pixels. This is only supported on Windows Phone.
logo_150x150	Logo image file of size 150x150 pixels. This is supported on all Windows Store App platforms.
logo_310x150	Logo image file of size 310x150 pixels. This is supported on all Windows Store App platforms.
logo_310x310	Logo image file of size 310x310 pixels. This is supported on all Windows Store App platforms.
Member	Description
logo_620x300	Splash screen image file of size 620x300 pixels. This is not supported on Windows Phone.
logo_480x800	Splash screen image file of size 480x800 pixels. This is only supported on Windows Phone.

logo_large	Large logo image file. This has to be 150x150 pixels. Supported on all Windows Store App platforms. Default provided by the mkspec.
logo_medium	Medium logo image file. For Windows Phone the image must have a pixel size of 71x71, for other Windows Store App platforms 70x70. Default provided by the mkspec.
logo_small	Small logo image file. For Windows Phone the image must have a pixel size of 44x44, for other Windows Store App platforms 30x30. Default provided by the mkspec.
logo_splash	Splash screen image file. For Windows Phone the image must have a pixel size of 480x800, for other Windows Store App platforms 620x300. Default provided by the mkspec.
logo_store	Logo image file for Windows Store. Default provided by the mkspec.
logo_wide	Wide logo image file. This has to be 310x150 pixels. Supported on all Windows Store App platforms. Default provided by the mkspec.
name	The name of the package as displayed to the user. Defaults to TARGET.
phone_product_id	The GUID of the product. Defaults to the value of <code>WINRT_MANIFEST.identity</code> . (Windows Phone only)
phone_publisher_id	The GUID of the publisher. Defaults to an invalid GUID. (Windows Phone only)
publisher	Display name of the publisher. Defaults to <code>Default publisher display name</code> .
publisher_id	The publisher's distinguished name (default: CN=MyCN).
target	The name of the target (.exe). Defaults to TARGET.
version	The version number of the package. Defaults to 1.0.0.0.
minVersion	The minimum required Windows version to run the package. Defaults to the environment variable <code>UCRTVersion</code> .
maxVersionTested	The maximum Windows version the package has been tested against. Defaults to <code>WINRT_MANIFEST.minVersion</code>

You can use any combination of those values.

For example:

```
WINRT_MANIFEST.publisher = MyCompany
WINRT_MANIFEST.logo_store = someImage.png
```

```
WINRT_MANIFEST.logo_store = SomeImage.png  
WINRT_MANIFEST.capabilities += internetClient  
WINRT_MANIFEST.capabilities_device += location
```

Additionally, an input manifest file can be specified by using `WINRT_MANIFEST`.

For example:

```
WINRT_MANIFEST = someManifest.xml.in
```

In case the input manifest file should not be processed and only copied to the target directory, the verbatim configuration needs to be set.

```
WINRT_MANIFEST = someManifest.xml.in  
WINRT_MANIFEST.CONFIG += verbatim
```

**Note:** The required image sizes of *logo\_small*, *logo\_medium*, and *logo\_large* depend on the target platform. The general descriptions are overwritten if a description that specifies the size is provided.

## 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
```

到达项目文件所在的路径，  
包含项目文件的名称。

`_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.

包含该项目文件的路径，类似于 `dirname`

`_PRO_FILE_PWD_`

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

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

```
message($_PRO_FILE_PWD_)
```

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

[< Reference](#)

[Replace Functions >](#)

## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Replace Functions](#)

## Contents



### Built-in Replace Functions

- `absolute_path(path[, base])`
- `basename(variablename)`
- `cat(filename[, mode])`
- `clean_path(path)`
- `dirname(file)`
- `enumerate_vars`
- `escape_expand(arg1 [, arg2 ..., argn])`
- `find(variablename, substr)`
- `files(pattern[, recursive=false])`
- `first(variablename)`
- `format_number(number[, options...])`
- `fromfile(filename, variablename)`
- `getenv(variablename)`
- `join(variablename, glue, before, after)`
- `last(variablename)`
- `list(arg1 [, arg2 ..., argn])`
- `lower(arg1 [, arg2 ..., argn])`
- `member(variablename [, start [, end]])`

num\_add(arg1 [, arg2 ..., argn])  
prompt(question [, decorate])  
quote(string)  
re\_escape(string)  
relative\_path(filePath[, base])  
replace(string, old\_string, new\_string)  
resolve\_depends(variablename, prefix)  
reverse(variablename)  
section(variablename, separator, begin, end)  
shadowed(path)  
shell\_path(path)  
shell\_quote(arg)  
size(variablename)  
sort\_depends(variablename, prefix)  
sorted(variablename)  
split(variablename, separator)  
sprintf(string, arguments...)  
str\_member(arg [, start [, end]])  
str\_size(arg)  
system(command[, mode[, stsvar]])  
system\_path(path)  
system\_quote(arg)  
take\_first(variablename)  
take\_last(variablename)  
unique(variablename)  
upper(arg1 [, arg2 ..., argn])  
val\_escape(variablename)

## Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

### Reference



- [All Qt C++ Classes](#)
- [All QML Types](#)
- [All Qt Modules](#)
- [Qt Creator Manual](#)
- [All Qt Reference Documentation](#)

### Getting Started

- [Getting Started with Qt](#)
- [What's New in Qt 5](#)
- [Examples and Tutorials](#)
- [Supported Platforms](#)
- [Qt Licensing](#)

### Overviews

- [Development Tools](#)
- [User Interfaces](#)
- [Core Internals](#)
- [Data Storage](#)
- [Multimedia](#)
- [Networking and Connectivity](#)
- [Graphics](#)



# Replace Functions

qmake provides functions for processing the contents of variables during the configuration process. These functions are called *replace functions*. Typically, they return values that you can assign to other variables. You can obtain these values by prefixing a function with the \$\$ operator. Replace functions can be divided into built-in functions and function libraries.

See also [Test Functions](#).

## Built-in Replace Functions

Basic replace functions are implemented as built-in functions.

`absolute_path(path[, base])`



返回绝对路径

Returns the absolute path of `path`.

If `base` is not specified, uses the current directory as the base directory. If it is a relative path, it is resolved relative to the current directory before use.

For example, the following call returns the string `"/home/johndoe/myproject/readme.txt"`:

```
message($$absolute_path("readme.txt", "/home/johndoe/myproject"))
```

See also `clean_path()`, `relative_path()`.

`basename(variablename)`

返回路径变量的basename

Returns the basename of the file specified in `variablename`.

For example:

```
FILE = /etc/passwd
FILENAME = $$basename(FILE) #passwd
```

`cat(filename[, mode])`

返回文件的内容

Returns the contents of `filename`. You can specify the following options for `mode`:

- › `blob` returns the entire contents of the file as one value
- › `lines` returns each line as a separate value (without line endings)
- › `true` (default value) and `false` return file contents as separate values, split according to `qmake` value list splitting rules (as in variable assignments). If `mode` is `false`, values that contain only a newline character are inserted into the list to indicate where line breaks were in the file.

`clean_path(path)`

Returns `path` with directory separators normalized (converted to `/`) and redundant ones removed, and `"."`s and `".."`s resolved (as far as possible). This function is a wrapper around `QDir::cleanPath`.

See also `absolute_path()`, `relative_path()`, `shell_path()`, `system_path()`.

`dirname(file)`

返回dirname

Returns the directory name part of the specified file. For example:

```
FILE = /etc/X11R6/XF86Config
DIRNAME = $$dirname(FILE) #!/etc/X11R6
```

enumerate\_vars

返回已经定义变量的列表

Returns a list of all defined variable names.

escape\_expand(arg1 [, arg2 ..., argn])

Accepts an arbitrary number of arguments. It expands the escape sequences `\n`, `\r`, `\t` for each argument and returns the arguments as a list.

**Note:** If you specify the string to expand literally, you need to escape the backslashes, as illustrated by the following code snippet:

```
message("First line$$escape_expand(\\n)Second line")
```

find(variablename, substr)

返回满足条件的变量的所有值

Returns all the values in `variablename` that match the regular expression `substr`.

```
MY_VAR = one two three four
MY_VAR2 = $$join(MY_VAR, " -L", -L) -Lfive
MY_VAR3 = $$member(MY_VAR, 2) $$find(MY_VAR, t.*)
```

MY\_VAR2 will contain '-Lone -Ltwo -Lthree -Lfour -Lfive', and MY\_VAR3 will contain 'three two three'.

## 返回满足条件的文件列表

`files(pattern[, recursive=false])`

Expands the specified wildcard pattern and returns a list of filenames. If recursive is true, this function descends into subdirectories.

## 返回第一个值

`first(variablename)`

Returns the first value of variablename.

For example, the following call returns `firstname`:

```
CONTACT = firstname middlename surname phone  
message($first(CONTACT))
```

See also `take_first()`, `last()`.

`format_number(number[, options...])`

Returns number in the format specified by options. You can specify the following options:

- › `ibase=n` sets the base of the input to n
- › `obase=n` sets the base of the output to n
- › `width=n` sets the minimum width of the output to n. If the output is shorter than width, it is padded with spaces
- › `zeropad` pads the output with zeroes instead of spaces
- › `padsign` prepends a space to positive values in the output
- › `always sign` prepends a plus sign to positive values in the output
- › `leftalign` places the padding to the right of the value in the output

Floating-point numbers are currently not supported.

For example, the following call converts the hexadecimal number BAD to 002989:

```
message($format_number(BAD, ibase=16 width=6 zeropad))
```


`fromfile(filename, variablename)`

Evaluates `filename` as a qmake project file and returns the value assigned to `variablename`.

See also `infile()`.

`getenv(variablename)`  **返回环境变量的值**

Returns the value of the environment variable `variablename`. This is mostly equivalent to the `$(variablename)` syntax. The `getenv` function, however, supports environment variables with parentheses in their name.

`join(variablename, glue, before, after)`  **连接操作**

Joins the value of `variablename` with `glue`. If this value is not empty, this function prefixes the value with `before` and suffixes it with `after`. `variablename` is the only required field, the others default to empty strings. If you need to encode spaces in `glue`, `before`, or `after`, you must quote them.

`last(variablename)`  **返回变量的最后一个名字**

Returns the last value of `variablename`.

For example, the following call returns `phone`:

```
CONTACT = firstname middlename surname phone
message($last(CONTACT))
```

```
message($?last($command))
```

See also `take_last()`, `first()`.

`list(arg1 [, arg2 ..., argn])`

生成去重的参数的列表

Takes an arbitrary number of arguments. It creates a uniquely named variable that contains a list of the arguments, and returns the name of that variable. You can use the variable to write a loop as illustrated by the following code snippet

```
for(var, $$list(foo bar baz)) {  
    ...  
}
```

instead of:

```
values = foo bar baz  
for(var, values) {  
    ...  
}
```

`lower(arg1 [, arg2 ..., argn])`

返回小写的参数的列表

Takes an arbitrary number of arguments and converts them to lower case.

See also `upper()`.

`member(variablename [, start [, end]])`

Returns the slice of the list value of `variablename` with the zero-based element indices between `start` and `end` (inclusive).

If `start` is not given, it defaults to zero. This usage is equivalent to `$$first(variablename)`.

If `end` is not given, it defaults to `start`. This usage represents simple array indexing, as exactly one element will be returned.

It is also possible to specify `start` and `end` in a single argument, with the numbers separated by two periods.

Negative numbers represent indices starting from the end of the list, with `-1` being the last element.

If either index is out of range, an empty list is returned.

If `end` is smaller than `start`, the elements are returned in reverse order.

**Note:** The fact that the end index is inclusive and unordered implies that an empty list will be returned only when an index is invalid (which is implied by the input variable being empty).

See also `str_member()`.

数字加法

`num_add(arg1 [, arg2 ..., argn])`

Takes an arbitrary number of numeric arguments and adds them up, returning the sum.

Subtraction is implicitly supported due to the possibility to simply prepend a minus sign to a numeric value to negate it:

```
sum = $$num_add($$first, -$$second)
```

If the operand may be already negative, another step is necessary to normalize the number:

```
second_neg = -$$second
second_neg =~ s/^--//
sum = $$num_add($$first, $$second_neg)
```

`prompt(question [, decorate])`

从标准输入读入一个变量

Displays the specified `question`, and returns a value read from `stdin`.

If `decorate` is *true* (the default), the question gets a generic prefix and suffix identifying it as a prompt.

`quote(string)`

Converts a whole `string` into a single entity and returns the result. This is just a fancy way of enclosing the string into double quotes.

`re_escape(string)`

Returns the `string` with every special regular expression character escaped with a backslash. This function is a wrapper around `QRegExp::escape`.

`relative_path(filePath[, base])`

返回相对路径

Returns the path to `filePath` relative to `base`.

If `base` is not specified, it is the current project directory. If it is relative, it is resolved relative to the current project directory before use.

If `filePath` is relative, it is first resolved against the base directory; in that case, this function effectively acts as `$$clean_path()`.

See also `absolute_path()`, `clean_path()`.

`replace(string, old_string, new_string)`

替换

Replaces each instance of `old_string` with `new_string` in the contents of the variable supplied as `string`. For example, the code

```
MESSAGE = This is a tent.
```



```
message($$replace(MESSAGE, tent, test))
```

prints the message:

```
This is a test.
```

## resolve\_depends(variablename, prefix)

This is an internal function that you will typically not need.

## reverse(variablename)

Returns the values of `variablename` in reverse order.

## section(variablename, separator, begin, end)

Returns a section of the value of `variablename`. This function is a wrapper around [QString::section](#).

For example, the following call outputs surname:

```
CONTACT = firstname:middlename:surname:phone  
message($$section(CONTACT, :, 2, 2))
```

## shadowed(path)

Maps the path from the project source directory to the build directory. This function returns `path` for in-source builds. It returns an empty string if `path` points outside of the source tree.

## shell\_path(path)

Converts all directory separators within `path` to separators that are compatible with the shell that is used while building the project (that is, the shell that is invoked by the make tool). For example, slashes are converted to backslashes when the Windows shell is used.

See also `system_path()`.

## shell\_quote(arg)

引用命令行参数

Quotes `arg` for the shell that is used while building the project.

See also `system_quote()`.

## size(variablename)

Returns the number of values of `variablename`.

See also `str_size()`.

## sort\_depends(variablename, prefix)

This is an internal function that you will typically not need.

## sorted(variablename)

Returns the list of values in `variablename` with entries sorted in ascending ASCII order.

Numerical sorting can be accomplished by zero-padding the values to a fixed length with the help of the `format_number()` function.

## split(variablename, separator)

分割

Splits the value of `var iablename` into separate values, and returns them as a list. This function is a wrapper around `QString::split`.

For example:

```
CONTACT = firstname:middlename:surname:phone  
message($$split(CONTACT, :))
```

## `sprintf(string, arguments...)`

Replaces `%1-%9` in `string` with the arguments passed in the comma-separated list of function arguments and returns the processed string.

## `str_member(arg [, start [, end]])`

This function is identical to `member()`, except that it operates on a string value instead of a list variable, and consequently the indices refer to character positions.

This function can be used to implement many common string slicing operations:

```
# $$left(VAR, len)  
left = $$str_member(VAR, 0, $$num_add($$len, -1))  
  
# $$right(VAR, len)  
right = $$str_member(VAR, -$num, -1)  
  
# $$mid(VAR, off, len)  
mid = $$str_member(VAR, $$off, $$num_add($$off, $$len, -1))  
  
# $$mid(VAR, off)  
mid = $$str_member(VAR, $$off, -1)  
  
# $$reverse(VAR)
```

```
reverse = $$str_member(VAR, -1, 0)
```

**Note:** In these implementations, a zero `len` argument needs to be handled separately.

See also [member\(\)](#), [num\\_add\(\)](#).

## str\_size(arg)

Returns the number of characters in the argument.

See also [size\(\)](#).

`system(command[, mode[, stsvar]])`

执行系统命令并可以使用标准输出的返回值

You can use this variant of the `system` function to obtain stdout from the command and assign it to a variable.

For example:

```
UNAME = $$system(uname -s)
contains( UNAME, [lL]inux ):message( This looks like Linux ($$UNAME) to me )
```

If you pass `stsvar`, the command's exit status will be stored in that variable. If the command crashes, the status will be -1, otherwise a non-negative exit code of the command's choosing. Usually, comparing the status with zero (success) is sufficient.

See also the test variant of [system\(\)](#).

## system\_path(path)

Converts all directory separators within `path` to separators that are compatible with the shell that is used by the `system()` functions to invoke commands. For example, slashes are converted to backslashes for the Windows shell.

See also [shell\\_path\(\)](#).

## system\_quote(arg)

Quotes `arg` for the shell that is used by the `system()` functions.

See also [shell\\_quote\(\)](#).

## take\_first(variablename)

Returns the first value of `variablename` and removes it from the source variable.

This provides convenience for implementing queues, for example.

See also [take\\_last\(\)](#), [first\(\)](#).

## take\_last(variablename)

Returns the last value of `variablename` and removes it from the source variable.

This provides convenience for implementing stacks, for example.

See also [take\\_first\(\)](#), [last\(\)](#).

## unique(variablename)



Returns the list of values in `variablename` with duplicate entries removed. For example:

```
ARGS = 1 2 3 2 5 1
ARGS = $$unique(ARGS) #1 2 3 5
```

转换为大写

`upper(arg1 [, arg2 ..., argn])`

Takes an arbitrary number of arguments and converts them to upper case.

See also `lower()`.

`val_escape(variablename)`

Escapes the values of `variablename` in a way that enables parsing them as qmake code.

[< Variables](#)

[Test Functions >](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.

#### Download

Start for Free  
Qt for Application Development  
Qt for Device Creation  
Qt Open Source  
Terms & Conditions  
Licensing FAQ

#### Product

Qt in Use  
Qt for Application Development  
Qt for Device Creation  
Commercial Features  
Qt Creator IDE  
Qt Quick

#### Services

Technology Evaluation  
Proof of Concept  
Design & Implementation  
Productization  
Qt Training  
Partner Network

#### Developers

Documentation  
Examples & Tutorials  
Development Tools  
Wiki  
Forums  
Contribute to Qt

#### About us

Training & Events  
Resource Center  
News  
Careers  
Locations  
Contact Us



[Sign In](#) [Feedback](#) © 2018 The Qt Company

# Qt Documentation

Google Search Documentation



Qt 5.11 > [qmake Manual](#) > [Test Functions](#)

## Contents



### Built-in Test Functions

`cache(variablename, [set|add|sub] [transient] [super|stash], [source variablename])`

`CONFIG(config)`

`contains(variablename, value)`

`count(variablename, number)`

`debug(level, message)`

`defined(name[, type])`

`equals(variablename, value)`

`error(string)`

`eval(string)`

`exists(filename)`

`export(variablename)`

`for(iterate, list)`

`greaterThan(variablename, value)`

`if(condition)`

`include(filename)`

`infile(filename, var, val)`

`isActiveConfig`

`isEmpty(variablename)`



isEqual  
lessThan(variablename, value)  
load(feature)  
log(message)  
message(string)  
mkpath(dirPath)  
requires(condition)  
system(command)  
touch(filename, reference\_filename)  
unset(variablename)  
versionAtLeast(variablename, versionNumber)  
versionAtMost(variablename, versionNumber)  
warning(string)  
write\_file(filename, [variablename, [mode]])

#### Test Function Library

packagesExist(packages)  
prepareRecursiveTarget(target)  
qtCompileTest(test)  
qtHaveModule(name)

#### Resource Center

Find webinars, use cases, tutorials, videos & more at [resources.qt.io](https://resources.qt.io)

#### Reference

[All Qt C++ Classes](#)

[All QML Types](#)



[All Qt Modules](#)  
[Qt Creator Manual](#)  
[All Qt Reference Documentation](#)

## Getting Started

[Getting Started with Qt](#)  
[What's New in Qt 5](#)  
[Examples and Tutorials](#)  
[Supported Platforms](#)  
[Qt Licensing](#)

## Overviews

[Development Tools](#)  
[User Interfaces](#)  
[Core Internals](#)  
[Data Storage](#)  
[Multimedia](#)  
[Networking and Connectivity](#)  
[Graphics](#)  
[Mobile APIs](#)  
[QML Applications](#)  
[All Qt Overviews](#)

# Test Functions

Test functions return a boolean value that you can test for in the conditional parts of scopes. Test functions can be divided into built-in functions and function libraries.

See also [Replace Functions](#).

条件函数

## Built-in Test Functions

Basic test functions are implemented as built-in functions.

`cache(variablename, [set|add|sub] [transient] [super|stash], [source variablename])`

This is an internal function that you will typically not need.

`CONFIG(config)` ← 检测变量CONFIG中是否包含某个值的内容

This function can be used to test for variables placed into the `CONFIG` variable. This is the same as scopes, but has the added advantage that a second parameter can be passed to test for the active config. As the order of values is important in `CONFIG` variables (that is, the last one set will be considered the active config for mutually exclusive values) a second parameter can be used to specify a set of values to consider. For example:

```
CONFIG = debug
CONFIG += release
CONFIG(release, debug|release):message(Release build!) #will print
CONFIG(debug, debug|release):message(Debug build!) #no print
```

Because release is considered the active setting (for feature parsing) it will be the `CONFIG` used to generate the build file. In the common case a second parameter is not needed, but for specific mutual exclusive tests it is invaluable.

`contains(variablename value)`

`contains(variablename, value)`

Succeeds if the variable `variablename` contains the value `value`; otherwise fails. It is possible to specify a regular expression for parameter `value`.

You can check the return value of this function using a scope.

**contains: 检查变量是否包含某个值**

For example:

```
contains( drivers, network ) {  
    # drivers contains 'network'  
    message( "Configuring for network build..." )  
    HEADERS += network.h  
    SOURCES += network.cpp  
}
```

The contents of the scope are only processed if the `drivers` variable contains the value `network`. If this is the case, the appropriate files are added to the `SOURCES` and `HEADERS` variables.

`count(variablename, number)`

Succeeds if the variable `variablename` contains a list with the specified number of values; otherwise fails.

This function is used to ensure that declarations inside a scope are only processed if the variable contains the correct number of values. For example:

```
options = $$find(CONFIG, "debug") $$find(CONFIG, "release")  
count(options, 2) {  
    message(Both release and debug specified.)  
}
```

`debug(level, message)`

Checks whether qmake runs at the specified debug level. If yes, it returns true and prints a debug message.

`defined(name[, type])` ← **检测某个变量或者函数是否定义**

**Tests whether the function or variable name is defined.** If type is omitted, checks all functions. To check only variables or particular type of functions, specify type. It can have the following values:

- › test only checks test functions
- › replace only checks replace functions
- › var only checks variables

`equals(variablename, value)` ← **相等性检测**

**Tests whether variablename equals the string value.**

For example:

```
TARGET = helloworld
equals(TARGET, "helloworld") {
    message("The target assignment was successful.")
}
```

`error(string)` ← **产生了不可恢复性的错误**

**This function never returns a value. qmake displays string as an error message to the user and exits. This function should only be used for unrecoverable errors.**

For example:

```
error(An error has occurred in the configuration process.)
```

`eval(string)` ← 求值

Evaluates the contents of the string using qmake syntax rules and returns true. Definitions and assignments can be used in the string to modify the values of existing variables or create new definitions.

For example:

```
eval(TARGET = myapp) {  
    message($$TARGET)  
}
```

**Note:** Quotation marks can be used to delimit the string, and the return value can be discarded if it is not needed.

`exists(filename)` ← 文件存在性检测

Tests whether a file with the given filename exists. If the file exists, the function succeeds; otherwise it fails. If a regular expression is specified for the filename, this function succeeds if any file matches the regular expression specified.

For example:

```
exists( $(QTDIR)/lib/libqt-mt* ) {  
    message( "Configuring for multi-threaded Qt..." )  
    CONFIG += thread  
}
```

**Note:** "/" should be used as a directory separator, regardless of the platform in use.

`export(variablename)`

Exports the current value of `variablename` from the local context of a function to the global context.

`for(iterate, list)` ← **循环迭代**

Starts a loop that iterates over all values in `list`, setting `iterate` to each value in turn. As a convenience, if `list` is `1..10` then `iterate` will iterate over the values 1 through 10.

For example:

```
LIST = 1 2 3
for(a, LIST):exists(file.$${a}):message(I see a file.$${a}!)
```

`greaterThan(variablename, value)` ← **大于比较**

Tests that the value of `variablename` is greater than `value`. First, this function attempts a numerical comparison. If at least one of the operands fails to convert, this function does a string comparison.

For example:

```
ANSWER = 42
greaterThan(ANSWER, 1) {
    message("The answer might be correct.")
}
```

It is impossible to compare two numbers as strings directly. As a workaround, construct temporary values with a non-numeric prefix and compare those.

these.

For example:

```
VALUE = 123
TMP_VALUE = x$$VALUE
greaterThan(TMP_VALUE, x456): message("Condition may be true.")
```

See also `lessThan()`.

`if(condition)` ← **条件检测**

Evaluates condition. It is used to group boolean expressions.

For example:

```
if(linux-g++*|macx-g++*):CONFIG(debug, debug|release) {
    message("We are on Linux or Mac OS, and we are in debug mode.")
}
```

`include(filename)` ← **包含文件**

Includes the contents of the file specified by `filename` into the current project at the point where it is included. This function succeeds if `filename` is included; otherwise it fails. The included file is processed immediately.

You can check whether the file was included by using this function as the condition for a scope. For example:

```
include( shared.pri )
OPTIONS = standard custom
```



```
!include( options.pri ) {  
    message( "No custom build options specified" )  
    OPTIONS -= custom  
}
```

## infile(filename, var, val)

Succeeds if the file `filename` (when parsed by qmake itself) contains the variable `var` with a value of `val`; otherwise fails. If you do not specify `val`, the function tests whether `var` has been assigned in the file.

## isActiveConfig

This is an alias for the `CONFIG` function.

## isEmpty(variablename) ← 检测变量是否为空

Succeeds if the variable `variablename` is empty; otherwise fails. This is the equivalent of `count( variablename, 0 )`.

For example:

```
isEmpty( CONFIG ) {  
    CONFIG += warn_on debug  
}
```

## isEqual

This is an alias for the `equals` function.

## lessThan(variablename value)

`lessThan(variableName, value)`

Tests that the value of `variableName` is less than `value`. Works as `greaterThan()`.

For example:

```
ANSWER = 42
lessThan(ANSWER, 1) {
    message("The answer might be wrong.")
}
```

`load(feature)` ← 加载特征文件

Loads the feature file (`.prf`) specified by `feature`, unless the feature has already been loaded.

`log(message)` ← 无前缀和换行的打印

Prints a message on the console. Unlike the `message` function, neither prepends text nor appends a line break.

See also `message()`.

`message(string)` ← 有前缀和换行的打印

Always succeeds, and displays `string` as a general message to the user. Unlike the `error()` function, this function allows processing to continue.

```
message( "This is a message" )
```

The above line causes "This is a message" to be written to the console. The use of quotation marks is optional, but recommended.

**Note:** By default, messages are written out for each Makefile generated by `gmake` for a given project. If you want to ensure that messages only

**NOTE:** By default, messages are written out for each invocation generated by qmake for a given project. If you want to ensure that messages only appear once for each project, test the `build_pass` variable **in conjunction with a scope** to filter out messages during builds. For example:

```
!build_pass:message( "This is a message" )
```

`mkpath(dirPath)`  **创建路径**

Creates the directory path `dirPath`. This function is a wrapper around the `QDir::makepath` function.

`requires(condition)`

Evaluates `condition`. If the condition is false, qmake skips this project (and its **SUBDIRS**) when building.

**Note:** You can also use the **REQUIRES** variable for this purpose. However, we recommend using this function, instead.

`system(command)`  **执行系统命令**

Executes the given command in a secondary shell. Succeeds if the command returns with a zero exit status; otherwise fails. You can check the return value of this function using a scope.

For example:

```
system("ls /bin"): HAS_BIN = TRUE
```

See also the replace variant of `system()`.

`touch(filename, reference_filename)`

Updates the time stamp of `filename` to match the time stamp of `reference_filename`.

Updates the time stamp of `filename` to match the time stamp of `reference_filename`.

## `unset(variablename)`

Removes `variablename` from the current context.

For example:

```
NARF = zort
unset(NARF)
!defined(NARF, var) {
  message("NARF is not defined.")
}
```

## `versionAtLeast(variablename, versionNumber)`

版本的最小检测

Tests that the version number from `variablename` is greater than or equal to `versionNumber`. The version number is considered to be a sequence of non-negative decimal numbers delimited by `:`; any non-numerical tail of the string will be ignored. Comparison is performed segment-wise from left to right; if one version is a prefix of the other, it is considered smaller.

## `versionAtMost(variablename, versionNumber)`

版本的最大检测

Tests that the version number from `variablename` is less than or equal to `versionNumber`. Works as `versionAtLeast()`.

## `warning(string)`

警告

Always succeeds, and displays `string` as a warning message to the user.

## `write_file(filename, [variablename, [mode]])`

Writes the values of `variablename` to a file with the name `filename`, each value on a separate line. If `variablename` is not specified, creates an

writes the values of `var TAB1ename` to a file with the name `FILEname`, each value on a separate line. If `var TAB1ename` is not specified, creates an empty file. If mode is `append` and the file already exists, appends to it instead of replacing it.

## Test Function Library

Complex test functions are implemented in a library of `.prf` files.

`packagesExist(packages)`  **库的检测**

Uses the `PKGCONFIG` mechanism to determine whether or not the given packages exist at the time of project parsing.

This can be useful to optionally enable or disable features. For example:

```
packagesExist(sqlite3 QtNetwork QtDeclarative) {  
    DEFINES += USE_FANCY_UI  
}
```

And then, in the code:

```
#ifdef USE_FANCY_UI  
    // Use the fancy UI, as we have extra packages available  
#endif
```

## `prepareRecursiveTarget(target)`

Facilitates the creation of project-wide targets similar to the `install` target by preparing a target that iterates through all subdirectories. For example:

```
TEMPLATE = subdirs
SUBDIRS = one two three
prepareRecursiveTarget(check)
```

Subdirs that have `have_no_default` or `no_<target>_target` specified in their `.CONFIG` are excluded from this target:

```
two.CONFIG += no_check_target
```

You must add the prepared target manually to `QMAKE_EXTRA_TARGETS`:

```
QMAKE_EXTRA_TARGETS += check
```

To make the target global, the code above needs to be included into every subdirs subproject. In addition, to make these targets do anything, non-subdirs subprojects need to include respective code. The easiest way to achieve this is creating a custom feature file. For example:

```
# <project root>/features/mycheck.prf
equals(TEMPLATE, subdirs) {
    prepareRecursiveTarget(check)
} else {
    check.commands = echo hello user
}
QMAKE_EXTRA_TARGETS += check
```

The feature file needs to be injected into each subproject, for example by `.qmake.conf`:

```
# <project root>/qmake.conf
CONFIG += mycheck
```

## qtCompileTest(test)

Builds a test project. If the test passes, true is returned and `config_<test>` is added to the `CONFIG` variable. Otherwise, false is returned.

To make this function available, you need to load the respective feature file:

```
# <project root>/project.pro
load(configure)
```

This also sets the variable `QMAKE_CONFIG_TESTS_DIR` to the `config.tests` subdirectory of the project's parent directory. It is possible to override this value after loading the feature file.

Inside the tests directory, there has to be one subdirectory per test that contains a simple qmake project. The following code snippet illustrates the .pro file of the project:

```
# <project root>/config.tests/test/test.pro
SOURCES = main.cpp
LIBS += -ltheFeature
# Note that the test project is built without Qt by default.
```

The following code snippet illustrates the main .cpp file of the project:

```
// <project root>/config.tests/test/main.cpp
```

```
#include <TheFeature/MainHeader.h>
int main() { return featureFunction(); }
```

The following code snippet shows the invocation of the test:

```
# <project root>/project.pro
qtCompileTest(test)
```

If the test project is built successfully, the test passes.

The test results are automatically cached, which also makes them available to all subprojects. It is therefore recommended to run all configuration tests in the top-level project file.

To suppress the re-use of cached results, pass `CONFIG+=recheck` to `qmake`.

See also [load\(\)](#).

## qtHaveModule(name)

Checks whether the Qt module specified by name is present. For a list of possible values, see [QT](#).

[< Replace Functions](#)

© 2018 The Qt Company Ltd. Documentation contributions included herein are the copyrights of their respective owners. The documentation provided herein is licensed under the terms of the [GNU Free Documentation License version 1.3](#) as published by the Free Software Foundation. Qt and respective logos are trademarks of The Qt Company Ltd. in Finland and/or other countries worldwide. All other trademarks are property of their respective owners.



## Download

[Start for Free](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Qt Open Source](#)  
[Terms & Conditions](#)  
[Licensing FAQ](#)

## Product

[Qt in Use](#)  
[Qt for Application Development](#)  
[Qt for Device Creation](#)  
[Commercial Features](#)  
[Qt Creator IDE](#)  
[Qt Quick](#)

## Services

[Technology Evaluation](#)  
[Proof of Concept](#)  
[Design & Implementation](#)  
[Productization](#)  
[Qt Training](#)  
[Partner Network](#)

## Developers

[Documentation](#)  
[Examples & Tutorials](#)  
[Development Tools](#)  
[Wiki](#)  
[Forums](#)  
[Contribute to Qt](#)

## About us

[Training & Events](#)  
[Resource Center](#)  
[News](#)  
[Careers](#)  
[Locations](#)  
[Contact Us](#)



[Sign In](#) [Feedback](#) © 2018 The Qt Company