

Contents

- [Adding New Configuration Features](#)
- [Installing Files](#)
- [Adding Custom Targets](#)
- [Adding Compilers](#)
- [Library Dependencies](#)

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:

- In a directory listed in the QMAKEFEATURES environment variable that contains a colon-separated list of directories.
- In a directory listed in the QMAKEFEATURES property variable that contains a colon-separated list of directories.
- 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 colon-separated list of directories. For example: \$QMAKEPATH/mkspecs/<features>.
- In a features directory residing beneath the directory provided by the QMAKESPEC environment variable. For example: \$QMAKESPEC/<features>.
- In a features directory residing in the data_install/mkspecs directory. For example: data_install/mkspecs/<features>.
- 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:

- features/unix, features/win32, or features/macx, depending on the platform in use
- 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:

- \$QMAKEFEATURES/myfeatures.prf (for each directory listed in the QMAKEFEATURES environment variable)
- \$\$QMAKEFEATURES/myfeatures.prf (for each directory listed in the QMAKEFEATURES property variable)
- myfeatures.prf (in the project's root directory)
- \$QMAKEPATH/mkspecs/features/unix/myfeatures.prf and \$QMAKEPATH/mkspecs/features/myfeatures.prf (for each directory listed in the QMAKEPATH environment variable)
- \$QMAKESPEC/features/unix/myfeatures.prf and \$QMAKESPEC/features/myfeatures.prf
- data_install/mkspecs/features/unix/myfeatures.prf and data_install/mkspecs/features/myfeatures.prf
- \$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/*
```

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
```

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
```

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
```

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
```

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 recursive 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 recursive is set in CONFIG. 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 \$(MAKE) -f Makefile.[subtarget] [recurse_target]. This member is used only when recursive is set in CONFIG.
target	The name of the custom build target.

Adding Compilers

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 CONFIG 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 TYPE_C, TYPE_UI, TYPE_QRC), 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 \$(QMAKE_COMP_VARNAME) when referred to in the pro file as \$(VARNAME).
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.

Option	Description
<code>target_predeps</code>	Indicates that the output should be added to the list of <code>PRE_TARGETDEPS</code> .
<code>explicit_dependencies</code>	The dependencies for the output only get generated from the <code>depends</code> member and from nowhere else.
<code>no_link</code>	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](#)

Copyright 2013 Digia Plc and/or it's subsidiaries. Documentation contributions included herein are the copyrights of their respective owners. Information about Qt licenses are available in the [Qt Licensing](#) page.

Hide Notes

Notes provided by the Qt Community

No notes



Community Notes are available under

[Attribution-Share Alike 2.5 Generic](#)