qmake User Guide

Qt 3.0

The definitive Qt documentation is provided in HTML format supplied with Qt, and available online at http://doc.trolltech.com. This PDF file was generated automatically from the HTML source as a convenience to users, although PDF is not an official Qt documentation format.

Contents

Introduction to qmake

Introduction to qmake

qmake is a tool created by Trolltech to write makefiles for different compilers and platforms.

Writing makefiles by hand can be difficult and error prone, especially if several makefiles are required for different compiler and platform combinations. With *qmake* developers create a simple single 'project' file, and run *qmake* to generate the appropriate makefiles. *qmake* takes care of all the compiler and platform dependencies, freeing developers to focus on their code. Trolltech use *qmake* as the primary build tool for the Qt library, and for the tools supplied with Qt.

qmake also takes care of Qt's special requirements, automatically including build rules for moc and *uic*.

Installing qmake

Installing qmake

qmake is built by default when Qt is built.

This section explains how to build *qmake* manually. Skip ahead to The 10 minute guide to using qmake, if you already have *qmake*.

Installing qmake manually

Before building Qt manually the following environment variables must be set:

- QMAKESPEC This must be set to the platform and compiler combination that you are using on your system. For example, if you are using Windows and Microsoft Visual Studio, you would set this environment variable to *win32-msvc*. If you are using Solaris and g++, you would set this environment variable to *solaris-g++*.
- QTDIR This must be set to where Qt is (or will be) installed. For example, *c:\qt* and *\local\qt*

Once the environment variables are set go into the qmake directory, *$QTDIR/qmake*, e.g. *C:\qt\qmake*. Now run *make* or *nmake* depending on your compiler.

When the make has completed, *qmake* is ready for use.

The 10 minute guide to using qmake

Creating a project file

qmake uses information stored in project (.pro) files to determine what should go in the makefiles it generates.

A basic project file contains information about the application, for example, which files are needed to compile the application, and which configuration settings to use.

Here's a simple example project file:

```
SOURCES = hello.cpp
HEADERS = hello.h
CONFIG += qt warn_on release
```

We'll provide a brief line-by-line explanation, deferring the detail until later on in the manual.

```
SOURCES = hello.cpp
```

This line specifies the source files that implement the application. In this case there is just one file, *hello.cpp*. Most applications require multiple files; this situation is dealt with by listing all the files on the same line space separated, like this:

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

Alternatively, each file can be listed on a separate line, by escaping the newlines, like this:

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

A more verbose approach is to list each file separately, like this:

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

This approach uses "+=" rather than "=" which is safer, because it always adds a new file to the existing list rather than replacing the list.

The HEADERS line is used to specify the header files created for use by the application, e.g.

```
HEADERS += hello.h
```

Any of the approaches used to list source files may be used for header files.

The CONFIG line is used to give *qmake* information about the application's configuration.

```
CONFIG += qt warn_on release
```

The "+=" is used here, because we add our configuration options to any that are already present. This is safer than using "=" which replaces all options with just those specified.

The *qt* part of the CONFIG line tells *qmake* that the application is built using Qt. This means that *qmake* will link against the Qt libraries when linking and add in the neccesary include paths for compiling.

The *warn_on* part of the CONFIG line tells *qmake* that it should set the compiler flags so that warnings are output.

The *release* part of the CONFIG line tells *qmake* that the application must be built as a release application. During development, programmers may prefer to replace *release* with *debug*, which is discussed later.

Project files are plain text (i.e. use an editor like notepad, vim or xemacs) and must be saved with a '.pro' extension. The name of the application's executable will be the same as the project file's name, but with an extension appropriate to the platform. For example, a project file called 'hello.pro' will produce 'hello.exe' on Windows and 'hello' on Unix.

Generating a makefile

When you have created your project file it is very easy to generate a makefile, all you need to do is go to where you have created your project file and type:

Makefiles are generated from the '.pro' files like this:

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

For Visual Studio users, *qmake* can also generate '.dsp' files, for example:

```
qmake -t vcapp -o hello.dsp hello.pro
```

qmake Tutorial

Introduction to the qmake tutorial

This tutorial teaches you how to use *qmake*. We recommend that you read the *qmake* manual after completing this tutorial.

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 *qt/qmake/example*. 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 *qt/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:

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

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

```
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 is HEADERS:

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

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

The target name is set automatically; it is the same as the project file, but with the suffix appropriate to 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
```

The final step is to set the *CONFIG* variable. Since this is a Qt application, we need to put 'qt' on the CONFIG line so that *qmake* will add the relevant libraries to be linked against and ensure that build lines for *moc* and *uic* are included in the makefile.

The finished project file should look like this:

```
CONFIG = qt
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 application directory, type:

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

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

Making an application debuggable

The release version of an application doesn't contain any debugging symbols or other debuggin 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 = qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```

Use *qmake* as before to generate a makefile and you will be able to debug your application.

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 - *hello_win.cpp* and *hello_x11.cpp*. We can't just add these to the *SOURCES* variable since this will put 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 *qmake* is run on.

A simple scope which will add in the platform dependent file for Windows looks like this:

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

So if *qmake* is run on Windows, it will add *hello_win.cpp* to the list of source files. If *qmake* is run on any other platform, it will simply ignore it. Now all that is left to be done is to create a scope for the x11 dependent file.

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

```
CONFIG = qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
```

Use *qmake* as before to generate a makefile.

Stopping qmake if a file doesn't exist

You may not want to create a makefile if a certain file doesn't 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. Simply replace the scope condition with the function. A check for a main.cpp file looks like this:

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

The "!" is used to negate the test, i.e. `exists( main.cpp )` is true if the file exists and `!exists( main.cpp )` is true if the file doesn't exist.

```
CONFIG = qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.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 the qDebug() statements when you run your application on the command line. Unless you build your application with the console setting, you won't see the output. We can easily put *console* on the CONFIG line so that on Windows the makefile will have this setting. But let's say that we

only want to add the CONFIG line if we are running on Windows *and* when *debug* is already on the CONFIG line. This requires using two nested scopes; just create one scope, then create the other inside that one. Put the settings to be processed inside the last scope, like this:

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

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

```
CONFIG = qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
    SOURCES += hello_win.cpp
}
x11 {
    SOURCES += hello_x11.cpp
}
!exists( main.cpp ) {
    error( "No main.cpp file found" )
}
win32:debug {
    CONFIG += console
}
```

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

qmake Concepts

Introducing qmake

qmake is an easy-to-use tool from Trolltech that creates makefiles for development projects across different platforms. *qmake* simplifies the generation of makefiles so that only a few lines of information are needed to create a makefile. *qmake* can be used for any software project whether it is written in Qt or not, although it also contains additional features to support Qt development.

qmake generates a makefile based on the information in a project file. Project files are created by the developer. Project files are usually simple, but can be quite sophisticated if required. *qmake* can also generate projects for Microsoft Visual studio without having to change the project file.

qmake's Concepts

The QMAKESPEC environment variable

Before *qmake* can be used to build makefiles, the QMAKESPEC environment variable must be set to the platform-compiler combination that is being used on the system. The QMAKESPEC environment variable tells qmake where to look to find platform and compiler specific information. This ensures that the right libraries are used, and that the generated makefile uses the correct syntax. A list of the currently supported platform-compiler combinations can be found in qt/mkspecs. Just set your environment variable to one of the directories listed.

For example, if you are using Microsoft Visual Studio on Windows, then you would set the QMAKESPEC environment variable to *win32-msvc*. If you are using gcc on Solaris then you would set your QMAKESPEC environment variable to *solaris-g++*.

Inside each of the directories in qt/mkspecs, there is a *qmake.conf* file which contains the platform and compiler specific information. These settings are applied to any project that is built using *qmake* and should not be modified unless you're an expert. For example, if all your applications had to link against a particular library, you might add this information to the relevant *qmake.conf* file.

Project (.pro) files

A project file is used to tell *qmake* the details it needs to know about creating a makefile for the application. For instance, a list of source files and header files that should be put into the project file; any application specific configuration, such as an extra library that should be linked against, or an extra include path.

Templates

The template variable tells *qmake* what sort of makefile should be generated for the application. The following choices are available:

- app - Creates a makefile that builds an application. This is the default, so if a template is not specified, this is used.
- lib - Creates a makefile that builds a library.
- vcapp - Creates a Visual Studio Project file which builds an application.
- vclib - Creates a Visual Studio Project file which builds a library.
- subdirs - This is a special template which creates a makefile which will go into the specified directories and create a makefile for the project file and call make on it.

The 'app' template

The 'app' template tells *qmake* to generate a makefile that will build an application. 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.

- HEADERS - A list of all the header files for the application.
- SOURCES - A list of all the source files for the application.
- FORMS - A list of all the .ui files (created using *Qt Designer*) for the application.
- LEXSOURCES - A list of all the lex source files for the application.
- YACCSOURCES - A list of all the 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.
- DEF_FILE - Windows only: A .def file to be linked against for the application.
- RC_FILE - Windows only: A resource file for the application.
- RES_FILE - Windows only: A resource file to be linked against for the application.

You only need to use the system variables that you have values for, for instance, if you don't have any extra INCLUDEPATHs then you don't need to specify any, *qmake* will add in the default ones needed. For instance, an example project file might look like this:

```
TEMPLATE = app
DESTDIR  = c:\helloapp
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
DEFINES += QT_DLL
CONFIG  += qt warn_on release
```

For items that are single valued, e.g. 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 item's value with the new value, for example if we wrote `DEFINES=QT_DLL`, all other definitions would be deleted.

The 'lib' template

The 'lib' template tells *qmake* to generate a makefile that will build a library. When using this template, in addition to the system variables mentioned above for the 'app' template the *VERSION* variable is supported. You should use these in your .pro file to specify information about the library.

- VERSION - The version number of the target library, for example, 2.3.1.

The 'subdirs' template

The 'subdirs' template tells qmake to generate a makefile that will go into the specified subdirectories and generate a makefile for the project file in the directory and call make on it.

The only system variable that is recognised for this template is the *SUBDIRS* variable. This variable contains a list of all the subdirectories that contain project files to be processed. It is essential that the project file in the sub directory has the same name as the subdirectory, so that *qmake* can find it. For example, if the subdirectory is called 'myapp' then the project file in that directory should be called *myapp.pro* in that directory.

The CONFIG variable

The config variable specifies the options that the compiler should use and the libraries that should be linked against. Anything can be added to the config variable, but the options covered below are recognised by qmake internally.

The following options control what compiler flags are used:

- release - The application is to be built in release mode. This is ignored if 'debug' is specified.
- debug - The application is to be built in debug mode.
- warn_on - The compiler should output as many warnings as possible. This is ignored if 'warn_off' is specified.
- warn_off - The compiler should output as few warnings as possible.

The following options define the type of library/application to be built:

- qt - The application is a Qt application and should link against the Qt library.
- thread - The application is a multi-threaded application.
- x11 - The application is an X11 application or library.
- windows - 'app' template only: the application is a Windows window application.
- console - 'app' template only: the application is a Windows console application.
- dll - 'lib' template only: The library is a shared library (dll).
- staticlib - 'lib' template only: The library is a static library.
- plugin - 'lib' template only: The library is a plugin; this enables the dll option.

For example, if your application uses the Qt library and you want to build it as a debuggable multi-threaded application, your project file will have the following line:

```
CONFIG += qt thread debug
```

Note, that you must use "+=", not "=", or *qmake* will not be able to use the settings used to build Qt as a guide as what type of Qt library was built.

qmake's Advanced Concepts

qmake's Advanced Concepts

The *qmake* project files we've seen up to now have been very simple, just a list of *name* = *value* and *name* += *value* lines. *qmake* provides a lot more power, for example you can use a single project file to produce makefiles for multiple platforms.

Operators

So far, you have seen the = operator and += operator being used in a project file. There are more operators available for use; but some of these should be used carefully as they may change more than you expect them to.

The '=' operator

This operator simply assigns a value to a variable, it is used like this:

```
TARGET = myapp
```

This sets the TARGET variable to *myapp*. This will remove any previously set TARGET.

The '+=' operator

This operator adds a value to the list of values in a variable. It is used like this:

```
DEFINES += QT_DLL
```

This adds QT_DLL to the list of pre-processor defines to be put in the makefile.

The '-=' operator

This operator removes a value from the list of values in a variable. It is used like this:

```
DEFINES -= QT_DLL
```

This removes QT_DLL from the list of pre-processor defines to be put in the makefile.

The '*=' operator

This operator only adds a value to the list of values in a variable if it doesn't already exist. It is used like this:

```
DEFINES *= QT_DLL
```

QT_DLL will only be added to the list of pre-processor defines if it is not already defined.

The '~=' operator

This operator replaces any values that match the regexp with the specified value. It is used like this:

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

This removes any values in the list that start with QT_D or QT_T with QT.

Scopes

A scope are similar to 'if' statements, if a certain condition is true, the settings inside the scope are processed. A scope is written like this:

```
win32 {
    DEFINES += QT_DLL
}
```

The above code will add the QT_DLL define to the makefile if *qmake* is used on a Windows platform. If *qmake* is used on a different platform than Windows, the define will be ignored.

For example, suppose we want to process something on all platforms *except* for Windows. We can achieve this by negating the scope like this:

```
!win32 {
    DEFINES += QT_DLL
}
```

Any entry on the CONFIG line is also a scope. For example, if you write this:

```
CONFIG += warn_on
```

you will have a scope called 'warn_on'. This makes it easy to change the configuration for a project without losing all the custom settings that might be needed for a specific configuration. Since it is possible to put your own values on the CONFIG line, this provides you with a very powerful configuration tool for your makefiles. For example:

```
CONFIG += qt warn_on debug
debug {
    TARGET = myappdebug
}
release {
    TARGET = myapp
}
```

In the above code, two scopes are created which depend on what is put on the CONFIG line. In the example, *debug* is on the config line, so the TARGET variable is set to *myappdebug*. If *release* was on the config line, then the TARGET variable would be set to *myapp*.

It is also possible to check for two things before processing some settings. For instance, if you want to check if the platform is Windows and that the thread configuration is set, you would write this:

```
win32 {
    thread {
        DEFINES += QT_THREAD_SUPPORT
    }
}
```

To save writing many nested scopes, you can nest scopes using a colon like this:

```
win32:thread {
    DEFINES += QT_THREAD_SUPPORT
}
```

Variables

The variables that we have encountered so far are system variables, such as *DEFINES*, *SOURCES* and *HEADERS*. It is possible for you to create your own variables so that you use them in scopes. It's easy to create your own variable; just name it and assign something to it. For example:

```
MY_VARIABLE = value
```

There are no restricitions on what you do to your own variables, as *qmake* will just ignore them unless it needs to look at them for 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.

Functions

qmake provides built-in functions that perform simple, yet powerful tasks.

contains(variablename, value)

If *value* is in the list of values stored in the variable called *variablename*, then the settings inside the scope will be processed. For example:

```
contains( CONFIG, thread ) {
    DEFINES += QT_THREAD_SUPPORT
}
```

If *thread* is in the list of values for the *CONFIG* variable, then QT_THREAD_SUPPORT will be added to the list of values in the *DEFINES* variable.

count(variablename, number)

If *number* matches the number of values stored in the variable called *variablename*, then the settings inside the scope will be processed. For example:

```
count( DEFINES, 5 ) {
    CONFIG += debug
}
```

error(string)

This function outputs the string given and then makes *qmake* exit. For example:

```
error( "An error has occured" )
```

The text "An error has occured" will be displayed on the console and *qmake* will exit.

exists(filename)

If the specified file exists, then the settings inside the scope will be processed. For example:

```
exists( /local/qt/qmake/main.cpp ) {
    SOURCES += main.cpp
}
```

If */local/qt/qmake/main.cpp* exists then main.cpp is added to the list of source files.

Note that "/" can be used as a directory separator regardless of the platform.

include(filename)

The contents of filename are included at this point in the project file, so any settings in the specified file will be processed. An example of this is:

```
include( myotherapp.pro )
```

Any settings in the *myotherapp.pro* project file are now processed.

isEmpty(variablename)

This is the equivalent of using count(variablename, 0). If the variable called *variablename* has no elements, then the settings inside the scope will be processed. An example of this is:

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

message(string)

This function simply outputs a message on the console.

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

The text "This is a message" is output to the console and processing of the project file carries on.

system(command)

The specified command is performed and if it returns an exit code of 1, the settings inside the scope are processed. For example:

```
system( ls /bin ) {
    SOURCES += bin/main.cpp
    HEADERS += bin/main.h
}
```

So if the command *ls /bin* returns 1 then *bin/main.cpp* is added to the list of sources and *bin/main.h* is added to the list of headers.

qmake Command Reference

qmake Command Reference

About This Reference

This reference is a detailed index of all command line options, configurations and internal variables used by the cross-platform makefile generation utility *qmake*.

Command Line Options

Syntax

```
qmake [options] files
```

Options

The following options can be specified on the command line to *qmake*:

- `-o` file *qmake* output will be directed to *file*. if this argument is not specified, then *qmake* will try to guess a suitable name. If '-' is specified, output is directed to stdout.
- `-unix` *qmake* will run in unix mode. In this mode, Unix file naming and path conventions will be used. This is the default mode on all Unices.
- `-win32` *qmake* will run in win32 mode. In this mode, Windows file naming and path conventions will be used. This is the default mode on Windows.
- `-d` *qmake* will output useful debugging information.
- `-t` tmpl *qmake* will override any set TEMPLATE variables with tmpl.
- `-help` *qmake* will go over these features and give some useful help.

qmake supports two different modes of operation. The first mode, which is the default is makefile generation. In this mode, *qmake* will take a .pro file and turn it into a makefile. Creating makefiles is covered by this reference guide, there is another mode which generates .pro files.

To toggle between these modes you must specify in the first argument what mode you want to use. If no mode is specified, *qmake* will assume you want makefile mode. The available modes are:

- `-makefile` *qmake* output will be a makefile (Makefile mode).
- `-project` *qmake* output will be a project file (Project file mode).

Makefile Mode

In Makefile mode *qmake* will generate a makefile. Additionally you may supply the following arguments in this mode:

- `-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 file found
- `-spec` spec *qmake* will use *spec* as a path to platform-compiler information and QMAKESPEC will be ignored.

The `files` argument can be a list of one or more project files, separated by spaces. You may also pass qmake assignments on the command line here and they will be processed before all files specified, for example:

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

This will generate a Makefile, from test.pro with Unix pathnames. However many of these arguments aren't necessary as they are the default. Therefore the line can be simplified on Unix to:

qmake "CONFIG+=test" test.pro

Projectfile Mode

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

- `-nopwd` *qmake* will not look in your current working directory for source code and only use the specified `files`

The `files` argument can be a list of files or directories. If a directory is specified, then 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 go into the correct variable depending on extension (i.e. .ui files go into FORMS, .cpp files go into SOURCES, etc).

System Variables

- Frequently Used System Variables
- Rarely Used System Variables

Frequently Used System Variables

The following variables are recognized by *qmake* and are used most frequently when creating project files.

CONFIG

The `CONFIG` variable specifies project configuration and compiler options. The values will be recognized internally by *qmake* and have special meaning. They are as follows.

These `CONFIG` values control compilation flags:

- release - Compile with optimization enabled, ignored if "debug" is specified
- debug - Compile with debug options enabled
- warn_on - The compiler should emit more warnings than normally, ignored if "warn_off" is specified
- warn_off - The compiler should only emit severe warnings.

These options define the application/library type:

- qt - The target is a Qt application/library and requires the Qt header files/library. The proper include and library paths for the Qt library will automatically be added to the project.
- opengl - The target requires the OpenGL (or Mesa) headers/libraries. The proper include and library paths for these libraries will automatically be added to the project.
- thread - The target is a multi-threaded application or library. The proper defines and compiler flags will automatically be added to the project.
- x11 - The target is a X11 application or library. The proper include paths and libraries will automatically be added to the project.
- 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.
- dll - The target is a shared object/DLL.The proper include paths, compiler flags and libraries will automatically be added to the project.
- staticlib - The target is a static library (lib only). The proper compiler flags will automatically be added to the project.
- plugin - The target is a plugin (lib only). This enables dll as well.

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

For example:

```
CONFIG += qt 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 QT_DLL
```

DEF_FILE

This is only used on Windows when using the 'app' template.

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

DESTDIR

Specifies where to put the target file.

For example:

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

DLLDESTDIR

Specifies where to copy the target dll.

HEADERS

Defines the header files for the project.

qmake will generate dependency information (unless -nodepend is specified on the command line) for the specified headers. *qmake* will also automatically detect if *moc* is required by the classes in these headers, and add 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.

INCLUDEPATH

This variable specifies the #include directories which should be searched when compiling the project. Use ';' or a space as the directory separator.

For example:

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

FORMS

This variable specifies the .ui files (see Qt Designer) to be processed through *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
```

LEXSOURCES

This variable contains 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

This variable contains a list of libraries to be linked into the project.

For example:

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

MOC_DIR

This variable specifies the directory where all intermediate moc files should be placed.

For example:

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

OBJECTS_DIR

This variable specifies the directory where all intermediate objects should be placed.

For example:

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

REQUIRES

This is a special variable processed by *qmake*. If the contents of this variable do not appear in CONFIG by the time this variable is assigned, then a minimal makefile will be generated that states what dependencies (the values assigned to REQUIRES) are missing.

This is mainly used in Qt's build system for building the examples.

SOURCES

This variable contains the name of all source files in the project.

For example:

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

See also HEADERS

SUBDIRS

This variable, when used with the 'subdir' TEMPLATE contains the names of all subdirectories to look for a project file.

For example:

```
SUBDIRS = kernel \
          tools
```

TARGET

This specifies the name of the target file.

For example:

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

The project file above would produce an executable named 'myapp' on unix and 'myapp.exe' on windows.

TEMPLATE

This variable contains the name of the template to use when generating the project. The allowed values are:

- app - Creates a makefile for building applications (the default)
- lib - Creates a makefile for building libraries
- subdirs - Creates a makefile for building targets in subdirectories
- vcapp - *win32 only* Creates an application project file
- vclib - *win32 only* Creates a library project file

For example:

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

VERSION

This variable contains the version number of the library if the 'lib' TEMPLATE is specified.

For example:

```
VERSION = 1.2.3
```

YACCSOURCES

This variable contains 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
```

Rarely Used System Variables

The following variables are also recognized by *qmake* but are either internal or very rarely used.

DESTDIR_TARGET

This variable is set internally by *qmake*, which is basically the DESTDIR variable with the TARGET variable appened at the end. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

DSP_TEMPLATE

This variable is set internally by *qmake*, which specifies where the dsp template file for basing generated dsp files is stored. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

LEXIMPLS

This variable contains 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

This variable contains the names of intermediate lex object files.The value of this variable is typically handled by *qmake* and rarely needs to be modified.

MAKEFILE

This variable specifies the name of the makefile which *qmake* should use when outputting the dependency information for building a project. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

MAKEFILE_GENERATOR

This variable contains 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.

OBJECTS

This variable is generated from the SOURCES variable. The extension of each source file will have been replaced by .o (Unix) or .obj (Win32). The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

OBJMOC

This variable is set by *qmake* if files can be found that contain the Q_OBJECT macro. `OBJMOC` contains the name of all intermediate moc object files. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

PRECOMPH

This variable contains a list of header files that require some sort of pre-compilation step (such as with moc). The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE

This variable contains 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

This variable contains the name of the *qmake* configuration to use when generating makefiles. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified. Use the QMAKESPEC environment variable instead.

QMAKE_AIX_SHLIB

If this variable is not empty, then this variable tells *qmake* to generate the TARGET as an AIX shared library.

QMAKE_APP_FLAG

This variable is empty unless the 'app' TEMPLATE is specified. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified. Use the following instead:

```
app {
        #conditional code for 'app' template here
}
```

QMAKE_APP_OR_DLL

This variable is empty unless the 'app' or 'dll' TEMPLATE is specified. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_AR_CMD

This is used on Unix platforms only

This variable contains the command for invoking the program which creates, modifies and extracts archives. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_DEBUG

This variable contains the flags for the C compiler in debug mode.The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_MT

This variable contains 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_MT_DBG

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

QMAKE_CFLAGS_MT_DLL

This is used on Windows only

This variable contains the compiler flags for creating a multi-threaded dll. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_MT_DLLDBG

This is used on Windows only

This variable contains the compiler flags for creating a debuggable multi-threaded dll. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_RELEASE

This variable contains the compiler flags for creating a non-debuggable application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CFLAGS_SHLIB

This is used on Unix platforms only

This variable contains 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

This variable contains 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

This variable is not empty if the warn_off TEMPLATE option is specified. 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 not empty if the warn_on TEMPLATE option is specified. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CLEAN

This variable contains any files which are not generated files (such as moc and uic generated files) and object files that should be removed when using "make clean".

QMAKE_CXXFLAGS_DEBUG

This variable contains the C++ compiler flags for creating a debuggable application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_MT

This variable contains 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_MT_DBG

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

QMAKE_CXXFLAGS_MT_DLL

`This is used on Windows only`

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

QMAKE_CXXFLAGS_MT_DLLDBG

`This is used on Windows only`

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

QMAKE_CXXFLAGS_RELEASE

This variable contains the C++ compiler flags for creating an application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_CXXFLAGS_SHLIB

This variable contains 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

This variable contains 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

This variable contains 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

This variable contains 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_EXTENSION_SHLIB

This variable contains the extention for shared libraries. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_FAILED_REQUIREMENTS

This variable contains the list of requirements that were failed to be met when *qmake* was used. For example, the sql module is needed and wasn't compiled into Qt. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_FILETAGS

This variable contains the file tags needed to be entered into the makefile, such as SOURCES and HEADERS. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_HPUX_SHLIB

This is used on Unix platforms only

If this variable is not empty then this variable tells *qmake* to generate the TARGET as an HPUX shared library. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_HPUX_SHLIBS

This is used on Unix platforms only

If this variable is not empty then this variable tells *qmake* to generate the TARGET as an HPUX shared library. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR

This variable contains the location of all known header files to be added to INCLUDEPATH when building an application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR_OPENGL

This variable contains the location of OpenGL header files to be added to INCLUDEPATH when building an application with OpenGL support. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR_QT

This variable contains the location of all known header file paths to be added to INCLUDEPATH when building a Qt application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR_THREAD

This variable contains the location of all known header file paths to be added to INCLUDEPATH when building a multi-threaded application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_INCDIR_X11

This is used on Unix platforms only

This variable contains the location of X11 header file paths to be added to INCLUDEPATH when building a X11 application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_CONSOLE

This is used on Windows only

This variable contains link flags when building console programs. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_CONSOLE_DLL

This is used on Windows only

This variable contains link flags when building console dlls. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_DEBUG

This variable contains link flags when building debuggable applications. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_PLUGIN

This variable contains link flags when building plugins. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_QT_DLL

This variable contains link flags when building programs that use the Qt library built as a dll. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_RELEASE

This variable contains link flags when building applications for release. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_SHAPP

This variable contains link flags when building applications which are using the 'app' template. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_SHLIB

This variable contains link flags when 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

This variable specifies 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

This variable contains link flags when 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

This is used on Windows only

This variable contains link flags when building windows projects. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LFLAGS_WINDOWS_DLL

This is used on Windows only

This variable contains link flags when building windows dll projects. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR

This variable contains the location of all known library directories.The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR_FLAGS

This is used on Unix platforms only

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

QMAKE_LIBDIR_OPENGL

This variable contains 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.

QMAKE_LIBDIR_QT

This variable contains the location of the Qt library directory.The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBDIR_X11

This is used on Unix platforms only

This variable contains 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

This variable contains all project libraries. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_CONSOLE

This is used on Windows only

This variable contains all project libraries that should be linked against when building a console application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_OPENGL

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

QMAKE_LIBS_OPENGL_QT

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

QMAKE_LIBS_QT

This variable contains all Qt libraries.The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_QT_DLL

This is used on Windows only

This variable contains all Qt libraries when Qt is built as a dll. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_QT_OPENGL

This variable contains all the libraries needed to link against if OpenGL support is turned on. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_QT_THREAD

This variable contains all the libraries needed to link against if thread support is turned on. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_RT

This is used with Borland compilers only

This variable contains the runtime library needed to link against when building an application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_RTMT

This is used with Borland compilers only

This variable contains the runtime library needed to link against when building a multi-threaded application. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_THREAD

This is used on Unix platforms only

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

QMAKE_LIBS_WINDOWS

This is used on Windows only

This variable contains all windows libraries.The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_LIBS_X11

This is used on Unix platforms only

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

QMAKE_LIBS_X11SM

This is used on Unix platforms only

This variable contains all X11 session management 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_SHLIB_CMD

This variable contains 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

This variable contains 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_MAKEFILE

This variable contains 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_MOC_SRC

This variable contains the names of all moc source files to generate and include in the project. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_QMAKE

This variable contains the location of qmake if it is not in the path. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_QT_DLL

This variable is not empty if Qt was built as a dll. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

QMAKE_RUN_CC

This variable 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

This variable 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

This variable 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

This variable 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_TARGET

This variable contains 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.

RC_FILE

This variable contains 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.

RES_FILE

This variable contains 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.

SRCMOC

This variable is set by *qmake* if files can be found that contain the Q_OBJECT macro. `SRCMOC` contains the name of all the generated moc files. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

TARGET_EXT

This variable specifies the target's extension. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

TARGET_x

This variable specifies the target's extension 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

This variable specifies the target's extension with version number. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

UICIMPLS

This variable contains a list of the generated implementation files by UIC. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

UICOBJECTS

This variable is generated from the UICIMPLS variable. The extension of each file will have been replaced by .o (Unix) or .obj (Win32). The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

VER_MAJ

This variable contains the major version number of the library, if the 'lib' template is specified.

VER_MIN

This variable contains the minor version number of the library, if the 'lib' template is specified.

VER_PAT

This variable contains the patch version number of the library, if the 'lib' template is specified.

YACCIMPLS

This variable contains a list of yacc source files. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

YACCOBJECTS

This variable contains a list of yacc object files. The value of this variable is typically handled by *qmake* or qmake.conf and rarely needs to be modified.

Functions

qmake recognizes the following functions:

include(filename)

This function will include the contents of *filename* into the current project at the point where was included. The function succeeds if *filename* was included, otherwise it fails. You can check the return value of this function using a scope.

For example:

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

exists(file)

This function will test if *file* exists. If the file exists, then it will succeed; otherwise it will fail. You can specify a regular expression in file and it will succeed if any file matches the regular expression specified.

For example:

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

contains(variablename, value)

This function will succeed if the variable *variablename* contains the value *value*. You can check the return value of this function using a scope.

For example:

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

```
        SOURCES += network.cpp
}
```

count(variablename, number)

This function will succeed if the variable *variablename* contains *number* elements, otherwise it will fail. You can check the return value of this function using a scope.

For example:

```
MYVAR = one two three
count( MYVAR, 3 ) {
        # always true
}
```

infile(filename, var, val)

This function will succeed if the file *filename* (when parsed by qmake itself) contains the variable *var* with a value of *val*. You may also not pass in a third argument (*val*) and the function will only test if *var* has been assigned to in the file.

isEmpty(variablename)

This function will succeed if the variable *variablename* is empty (same as `count(variable, 0)`).

system(command)

This function will execute `command` in a secondary shell and will succeed if the command exits with an exit status of 1. You can check the return value of this function using a scope.

For example:

```
system(ls /bin):HAS_BIN=FALSE
```

message(string)

This function will always succeed, and will display the given *string* to the user.

error(string)

This function will never return a value. It will display the given *string* to the user, and then exit *qmake*. This function should only be used for very fatal configurations.

For example:

```
release:debug:error(You can't have release and debug at the same time!)
```

Environment Variables and Configuration

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 QTDIR environment variable.

Note: the QMAKESPEC path will automatically be added to the INCLUDEPATH system variable.

Cache File

The cache file (mentioned above in the options) is a special file *qmake* will read to find settings not specified in the `qmake.conf` file, the .pro file, or the command line. If neither `-path` or `-nocache` are specified, *qmake* will try to find a file called `.qmake.cache` in parent directories. If it fails to find this file, it will silently ignore this step of processing.