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.

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.

This function was introduced in Qt 5.0.

CONFIG(config)

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)

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( 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])

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)

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

For example:

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

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)

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)

Tests whether a file with the given `filename` exists. If the file exists, the function succeeds; otherwise it fails.

The `filename` argument may contain wildcards. In that case, this function succeeds if any file matches.

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)

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.

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

Loops can be interrupted with `break()`. The `next()` statement skips the remainder of the loop's body and continues execution with the next iteration.

greaterThan(variablename, value)

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 these.

For example:

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

See also lessThan().

if(condition)

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)

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.

```
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)

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)

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)

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

This function was introduced in Qt 5.0.

See also message().

message(string)

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 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)

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

This function was introduced in Qt 5.0.

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)

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().

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

This function was introduced in Qt 5.0.

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)

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.

This function was introduced in Qt 5.10.

versionAtMost(variablename, versionNumber)

Tests that the version number from `variablename` is less than or equal to `versionNumber`. Works as versionAtLeast().

This function was introduced in Qt 5.10.

warning(string)

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 empty file. If `mode` is `append` and the file already exists, appends to it instead of replacing it.

This function was introduced in Qt 5.0.

Test Function Library

Complex test functions are implemented in a library of .prf files.

packagesExist(packages)

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
```

The feature file needs to be injected into each subproject, for example by .qmake.conf:

```
# <project root>/.qmake.conf
CONFIG += mycheck
```

This function was introduced in Qt 5.0.

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)
```

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().

This function was introduced in Qt 5.0.

qtHaveModule(name)

Checks whether the Qt module specified by `name` is present. For a list of possible values, see QT.

This function was introduced in Qt 5.0.1.

‹ Replace Functions

Qt DOCUMENTATION

Forum

Wiki

Downloads

Marketplace

Feedback Sign In