The SCons qt4 tool

Dirk Baechle

2010-11-18

Table of Contents

1. Basics

This tool can be used to compile Qt projects, designed for versions 4.x.y and higher. It is not usable for Qt3 and older versions, since some of the helper tools (`moc`, `uic`) behave different.

For activating the tool "qt4", you have to add its name to the Environment constructor, like this

```
env = Environment(tools=['default','qt4'])
```

On its startup, the Qt4 tool tries to read the variable `QT4DIR` from the current Environment and `os.environ`. If it is not set, the value of `QTDIR` (in Environment/`os.environ`) is used as a fallback.

So, you either have to explicitly give the path of your Qt4 installation to the Environment with

```
env['QT4DIR'] = '/usr/local/Trolltech/Qt-4.2.3'
```

or set the `QT4DIR` as environment variable in your shell.

Under Linux, "qt4" uses the system tool `pkg-config` for automatically setting the required compile and link flags of the single Qt4 modules (like QtCore, QtGui,...). This means that

1. you should have `pkg-config` installed, and
2. you additionally have to set `PKG_CONFIG_PATH` in your shell environment, such that it points to $`QT4DIR/lib/pkgconfig` (or $`QT4DIR/lib` for some older versions).

Based on these two environment variables (`QT4DIR` and `PKG_CONFIG_PATH`), the "qt4" tool initializes all `QT4_*` construction variables listed in the Reference manual. This happens when the tool is "detected" during Environment construction. As a consequence, the setup of the tool gets a two-stage process, if you want to override the values provided by your current shell settings:

```
# Stage 1: create plain environment
qtEnv = Environment()
# Set new vars
qtEnv['QT4DIR'] = '/usr/local/Trolltech/Qt-4.2.3
qtEnv['ENV']['PKG_CONFIG_PATH'] = '/usr/local/Trolltech/Qt-4.2.3/lib/pkgconfig'
# Stage 2: add qt4 tool
qtEnv.Tool('qt4')
```

2. Suggested boilerplate

Based on the requirements above, we suggest a simple ready-to-go setup as follows:

SConstruct

```
# Detect Qt version
qtdir = detectLatestQtDir()

# Create base environment
baseEnv = Environment()
#...further customization of base env

# Clone Qt environment
qtEnv = baseEnv.Clone()
# Set QT4DIR and PKG_CONFIG_PATH
qtEnv['ENV']['PKG_CONFIG_PATH'] = os.path.join(qtdir, 'lib/pkgconfig')
qtEnv['QT4DIR'] = qtdir
# Add qt4 tool
qtEnv.Tool('qt4')
#...further customization of qt env

# Export environments
Export('baseEnv qtEnv')

# Your other stuff...
# ...including the call to your SConscripts
```

In a SConscript

```
# Get the Qt4 environment
Import('qtEnv')
# Clone it
env = qtEnv.clone()
# Patch it
env.Append(CCFLAGS=['-m32']) # or whatever
# Use it
env.StaticLibrary('foo', Glob('*.cpp'))
```

The detection of the Qt directory could be as simple as directly assigning a fixed path

```
def detectLatestQtDir():
    return "/usr/local/qt4.3.2"
```

or a little more sophisticated

```
# Tries to detect the path to the installation of Qt with
# the highest version number
def detectLatestQtDir():
    if sys.platform.startswith("linux"):
        # Simple check: inspect only '/usr/local/Trolltech'
        paths = glob.glob('/usr/local/Trolltech/*')
        if len(paths):
            paths.sort()
```

```
            return paths[-1]
        else:
            return ""
    else:
        # Simple check: inspect only 'C:\Qt'
        paths = glob.glob('C:\\Qt\\*')
        if len(paths):
            paths.sort()
            return paths[-1]
        else:
            return os.environ.get("QTDIR","")
```

3. A first project

The following SConscript is for a simple project with some cxx files, using the QtCore, QtGui and QtNetwork modules:

```
Import('qtEnv')
env = qtEnv.Clone()
env.EnableQt4Modules([
                      'QtGui',
                      'QtCore',
                      'QtNetwork'
                     ])
# Add your CCFLAGS and CPPPATHs to env here...

env.Program('foo', Glob('*.cpp'))
```

4. MOC it up

For the basic support of automocing, nothing needs to be done by the user. The tool usually detects the `Q_OBJECT` macro and calls the "`moc`" executable accordingly.

If you don't want this, you can switch off the automocing by a

```
env['QT4_AUTOSCAN'] = 0
```

in your SConscript file. Then, you have to moc your files explicitly, using the Moc4 builder.

You can also switch to an extended automoc strategy with

```
env['QT4_AUTOSCAN_STRATEGY'] = 1
```

Please read the description of the `QT4_AUTOSCAN_STRATEGY` variable in the Reference manual for details.

For debugging purposes, you can set the variable `QT4_DEBUG` with

```
env['QT4_DEBUG'] = 1
```

which outputs a lot of messages during automocing.

5. Forms (.ui)

The header files with setup code for your GUI classes, are not compiled automatically from your `.ui` files. You always have to call the Uic4 builder explicitly like

```
env.Uic4(Glob('*.ui'))
env.Program('foo', Glob('*.cpp'))
```

6. Resource files (.qrc)

Resource files are not built automatically, you always have to add the names of the `.qrc` files to the source list for your program or library:

```
env.Program('foo', Glob('*.cpp')+Glob('*.qrc'))
```

For each of the Resource input files, its prefix defines the name of the resulting resource. An appropriate "`-name`" option is added to the call of the `rcc` executable by default.

You can also call the Qrc4 builder explicitly as

```
qrccc = env.Qrc4('foo') # ['foo.qrc'] -> ['qrc_foo.cc']
```

or (overriding the default suffix)

```
qrccc = env.Qrc4('myprefix_foo.cxx','foo.qrc') # -> ['qrc_myprefix_foo.cxx']
```

and then add the resulting cxx file to the sources of your Program/Library:

```
env.Program('foo', Glob('*.cpp') + qrccc)
```

7. Translation files

The update of the `.ts` files and the conversion to binary `.qm` files is not done automatically. You have to call the corresponding builders on your own.

Example for updating a translation file:

```
env.Ts4('foo.ts','.') # -> ['foo.ts']
```

By default, the `.ts` files are treated as *precious* targets. This means that they are not removed prior to a rebuild, but simply get updated. Additionally, they do not get cleaned on a "`scons -c`". If you want to delete the translation files on the "`-c`" SCons command, you can set the variable "`QT4_CLEAN_TS`" like this

```
env['QT4_CLEAN_TS']=1
```

Example for releasing a translation file, i.e. compiling it to a `.qm` binary file:

```
env.Qm4('foo') # ['foo.ts'] -> ['foo.qm']
```

or (overriding the output prefix)

```
env.Qm4('myprefix','foo') # ['foo.ts'] -> ['myprefix.qm']
```

As an extension both, the Ts4() and Qm4 builder, support the definition of multiple targets. So, calling

```
env.Ts4(['app_en','app_de'], Glob('*.cpp'))
```

and

```
env.Qm4(['app','copy'], Glob('*.ts'))
```

should work fine.

Finally, two short notes about the support of directories for the Ts4() builder. You can pass an arbitrary mix of cxx files and subdirs to it, as in

```
env.Ts4('app_en',['sub1','appwindow.cpp','main.cpp']))
```

where `sub1` is a folder that gets scanned recursively for cxx files by `lupdate`. But like this, you lose all dependency information for the subdir, i.e. if a file inside the folder changes, the .ts file is not updated automatically! In this case you should tell SCons to always update the target:

```
ts = env.Ts4('app_en',['sub1','appwindow.cpp','main.cpp'])
env.AlwaysBuild(ts)
```

Last note: specifying the current folder "`.`" as input to Ts4() and storing the resulting .ts file in the same directory, leads to a dependency cycle! You then have to store the .ts and .qm files outside of the current folder, or use `Glob('*.cpp'))` instead.