Deploying a Qt Application on Symbian

Qt uses information in the application's project file (`.pro`) to create installation files in Symbian's native `.sis` file format. The default installation packages are suitable for installation during development, but may need to be modified for commercial deployment.

This article explains the Qt packaging toolchain, and how to modify the Qt project file to customize the installation. It includes an explanation of the minimal changes that will be needed by most applications for commercial distribution, through to how to create a fully multilingual installation.

SIS installation files must also be digitally signed prior to installation. Signing is covered in detail in Qt & Application Signing.

What can you do with the Symbian packaging mechanism?

The packaging and installation mechanism is fairly flexible - it has had to be in order to distribute applications to hundreds of millions of phones in many different countries and languages. A small subset of the things you can do are:

- Display licence text on installation/removal
- Display prompts to the user during installation/removal
- Specify vendor information
- Run applications or documents during installation/removal
- Shut other applications down
- Install files to specific locations on the device
- Conditionally install files based on information from the target device (e.g. screen size, local language, presence of specific files or applications, device model, responses to prompts, etc.)
- Install only if certain libraries are on the device
- Install only on devices that are compatible
- Abort an installation under user defined conditions
- Patch or upgrade existing installations
- Display all user prompts and install all files based on current locale

Qt developers don't need to know very much about the underlying packaging toolchain. They do however need to understand the Package File Format in order to get the most out of the installation mechanism!

Toolchain overview

Qt *uses* the standard Symbian packaging toolchain to create and sign SIS files. Actually, it uses it twice; once to build and sign an installation file that contains the application itself, and then again to package the application installation file along with the Nokia Smart Installer for Symbian.

File:SymbianQtInstallationToolchainDiagram
Qt Packaging Process

You don't need to know very much about the Symbian toolchain because the entire process is wrapped up for you in a number of Qt make file calls, as shown below (using the command line):

`qmake`	Creates Symbian package definition files (*application_template.pkg* and *application_installer.pkg*) from the application's `.pro` file.
`make release-gcce`	Build application release version with the GCCE compiler
`make sis`	Creates `.sis` using *application_template.pkg* and signs it. The sis file contains the application only. This is suitable for deployment during development, where you can ensure that Qt is already present on the device.
`make installer_sis`	Creates `.sis` using *application_installer.pkg* and signs it. The installer sis file contains both the application `.sis` file and the smart installer.

Note that the commands above self-sign the SIS files; you will specify different certificates to sign the file for distribution, as discussed in Qt & Application Signing.

The discussion above assumes use of the command line. At time of writing Qt Creator cannot be used for creating Smart installer

based application installations, but is able to create the *application_template.pkg* based SIS files suitable for development.

Modifying the installation

The installation may be customised using the *qmake* DEPLOYMENT variable in the project file (within Symbian scope). This section provides an overview of what you can do with this variable; the following sections provide recipes to solve specific deployment problems.

Raw PKG file content can be specified using either *pkg_prerules* or *pkg_postrules* deployment variables. The *pkg_prerules* are injected into the package file before the package-body while *pkg_postrules* are injected afterwards.

Note: There are some restrictions on the format of data in the `.pro` file which mean that you can't just copy-paste in the desired `.pkg` file syntax:

- The `.pro` file lines are surrounded by double quotation marks. Use a backslash to *escape* quotation marks used in the package definition
- The `.pro` file uses the # character for comments. To use a # in your package definition use `$${LITERAL_HASH}`

The *pkg_prerules* are used to replace the default languages, package header (application name, UID, version, installation type) and vendor id. For example, to change the vendor information:

- ```
  vendorinfo = \
       "%{\"Vendor name in locale specific form\"}" \
       ":\"Vendor name in global form\""

  my_deployment.pkg_prerules = vendorinfo
  DEPLOYMENT += my_deployment
  ```

The *pkg_postrules* are used to add new dependencies, additional files, and any other required behaviour. For example

- Add a new hardware dependency:

  ```
  new_platform = \
       "; Some new platform" \
       "[0x1028315F],0,0,0,{\"S60ProductID\"}"

   my_deployment.pkg_prerules += new_platform
   DEPLOYMENT += my_deployment
  ```

- Add an addition file:

  ```
  addFiles.pkg_postrules =
  "\"$(EPOCROOT)\\epoc32\\release\\$(PLATFORM)\\$(TARGET)\\mybig.dll\" -
  \"!:\\sys\\bin\\mybig_installed.dll\""
  DEPLOYMENT += addFiles
  ```

The default platform and package dependencies are defined in the *default_deployment* item, and can be removed as shown below:

```
default_deployment.pkg_prerules -= pkg_platform_dependencies
default_deployment.pkg_prerules -= pkg_depends_webkit
default_deployment.pkg_prerules -= pkg_depends_qt
```

Deployment *sources* and *paths* can also be used to deploy new files, albeit less flexibly:

```
addFiles.sources = mybig.dll
```
```

```
addFiles.path = /sys/bin
DEPLOYMENT += addFiles
```

The changes described above all affect the *application_template.pkg*. The only changes you can make to the smart installer wrapper are to define a new package header:

```
DEPLOYMENT.installer_header = "$${LITERAL_HASH}{\"My Application
Installer\"},(0x12345678),1,2,3"
```

If you specify just UID of the installer package as the value, then installer package name and version will be autogenerated:

```
DEPLOYMENT.installer_header = 0x12345678
```

Changes needed for Symbian Signed

The default applications created by Qt are suitable for development. In order to deploy an application through Symbian Signed, a number of minimal changes need to be made:

- Change the UID to a value allocated by Symbian Signed
- Specify a global vendor name
- Update the version/build number

You may also need to change the caption name to be different to the executable name.

This section shows how to make these *minimal* changes as a difference against the default project file generated by the Qt Creator application wizard. Additions the the project file are marked with >>, removals are marked with <<, and comments are marked with *#NOTE*:

```
QT       += core gui

TARGET = QtHelloWorld
TEMPLATE = app;

SOURCES += main.cpp\
        qhelloworld.cpp
HEADERS  += qhelloworld.h
FORMS    += qhelloworld.ui

#NOTE: Remove lines below IF your application is not dependent on the Qt Mobility APIs
(otherwise leave)
<< CONFIG += mobility
<< MOBILITY =


symbian {

#NOTE: Replace UID with value allocated by Symbian Signed. Should be in 0x2 range for
Symbian Signing or 0xA range for self signing
<<    TARGET.UID3 = 0xef4329e5
>>    TARGET.UID3 = 0x2???????


    TARGET.EPOCSTACKSIZE = 0x14000
    TARGET.EPOCHEAPSIZE = 0x020000 0x800000
```

```
#NOTE: Update your version number with each build.
#NOTE: You may also need a different name than the executable for your caption.
>>    packageheader = "$${LITERAL_HASH}{\"Qt Hello World \"}, (0x2???????), 1, 2, 3,
TYPE=SA"

#NOTE: Add a vendor (company) names.
#NOTE: The global name is mandatory for Symbian Signed and should match the name in your
Symbian Signed account.
>>    vendorinfo = \
>>     "%{\"Vendor name in locale specific form\"}" \
>>     ":\"Vendor name in global form\""

#NOTE: The new new package header and vendor information are defined but not yet used
#NOTE: Add the variables to ''my_deployment.pkg_prerules'', and add this to the
DEPLOYMENT
>> my_deployment.pkg_prerules = packageheader vendorinfo
>> DEPLOYMENT += my_deployment
}
```

Change the language of the installation

The languages and dialects supported by an installation are specified in the Languages supported line, using language codes. The installer will display prompts in the best match of the languages in the installation file to the current phone language.

By default, Qt specifies that the installation supports a single language (English). To change this for French you would do:

```
# Define language header
languages = "&FR"
my_deployment.pkg_prerules = languages
DEPLOYMENT += my_deployment
```

Supporting *multiple* languages is more complicated, because it requires that you provide a translation for every single user visible string - including those in the package header, localised vendor ID, any files displayed during installation etc. This process is discussed in a following section

Update the package name, UID, version number, and type

The name of a SIS file, its UID, version number, and installation type are all part of the Package-Header (see link for full syntax). The following fragment shows how to set the header for both the application and installer packages:

```
# Define package header for application package
packageheader = "$${LITERAL_HASH}{\"My Application name\"}, (0x2theUID), 1, 2, 3,
TYPE=SA"

my_deployment.pkg_prerules = packageheader
DEPLOYMENT += my_deployment

# Define package header for smart installer installer package
DEPLOYMENT.installer_header = "$${LITERAL_HASH}{\"My Application
Installer\"},(0x2anotherUID),1,2,3"
```

Notes:

- Change the UIDs to values allocated by Symbian Signed
- Iterate the version number (major, minor, build) every time the application is signed

    - The application and installer version numbers should match
- The installation TYPE of "SA" (standard application) is the default type, and may be omitted

Adding the vendor information

There are two types of Vendor information (see link for package syntax):

- Localised vendor information is the vendor/company name in each language supported by the file
- The global (non localised) vendor name i used to identify the vendor for package updates. This is mandatory for Symbian Signing the application

The example below shows how you change both global and local vendor ids for a *single language* installation (the example also shows how to set the package header at the same time!). Note that changing the field in the `.pro` file automatically changes it in both the application and installer wrapper pkg files.

```
packageheader = "$${LITERAL_HASH}{\"MyApp-EN\"}, (0x2???????), 1, 2, 3, TYPE=SA"

vendorinfo = \
"%{\"Your Localised Vendor Name \"}" \
":\"Your Global Vendor Name \""

my_deployment.pkg_prerules = packageheader vendorinfo
DEPLOYMENT += my_deployment
```

Modifying package and platform dependencies

Package files can define two types of dependencies:

- *Component dependencies:* An installation file will only install if all the specified components are present on the target device
- *Platform dependencies:* An installation file will install if *any* of the specified platforms are present, as it is considered "compatible" with the platform

The syntax used is defined here: Dependency. The syntax is much the same for both types of dependency - the only difference being that square brackets are used for platform dependencies.

By default the *application_template.pkg* file defines a component dependency on the Qt SIS file in the appropriate version and a hardware dependency for all products from S60 3rd Edition, FP1. This ensures that by default the application will install on any product/platform that supports Qt, provided Qt is installed first. The default pkg file syntax is shown below:

```
; Default component dependency to Qt libraries
(0x2001E61C), 4, 6, 3, {"Qt"}
; Default HW/platform dependencies
[0x101F7961],0,0,0,{"S60ProductID"}
[0x102032BE],0,0,0,{"S60ProductID"}
[0x102752AE],0,0,0,{"S60ProductID"}
[0x1028315F],0,0,0,{"S60ProductID"}
; Default dependency to QtMobility libraries
(0x2002AC89), 1, 0, 1, {"QtMobility"}
```

Note that the dependency for Qt Mobility is present while the following line in in your project file.

```
CONFIG += mobility
```

The smart installer package will have the same *platform* dependencies as the application package, but will not be dependent on Qt, which it delivers. This is why the smart installer SIS file installs first - only after it completes is the application SIS file (which IS dependent on Qt) installed.

You might want to change the dependencies if, for example, your application can only run on Symbian platform, and not earlier S60 devices. In this case you can use the *default_deployment* variable to remove all the existing platform dependencies, and then

define your own deployment variable for the platforms that are supported:

```
# Remove all the existing platform dependencies
default_deployment.pkg_prerules -= pkg_platform_dependencies

#Add a dependency for just the S60 5th edition (Symbian^1) and later phones
supported_platforms = \
"; Application that only supports S60 5th edition" \
"[0x1028315F],0,0,0,{\"S60ProductID\"}"

my_deployment.pkg_prerules += supported_platforms
DEPLOYMENT += my_deployment
```

If necessary, component and other dependencies can also be removed using *default_deployment* as shown below:

```
#remove all dependencies
default_deployment.pkg_prerules=
#remove platform dependencies only
default_deployment.pkg_prerules -= pkg_platform_dependencies
#remove webkit component dependencies
default_deployment.pkg_prerules -= pkg_depends_webkit
#remove qt component dependency
default_deployment.pkg_prerules -= pkg_depends_qt
```

Add files to the installation

Qt automatically adds the files needed by most applications to the installation - including the application's executable and registration files. It is possible to deploy additional files, for example shared library DLLs, translation files, data files, image files etc. It is also possible to add files that are displayed or run during installation or uninstallation. There is a slightly different syntax for installing files that are installed irrespective of language and those that are language dependent. Both are documented here: Installation files.

If you just want to install a language neutral file then you can define a new DEPLOYMENT variable in the `.pro` file using sources and path.

```
symbian: {
addFiles.sources = myfile.dll
addFiles.path = /sys/bin
DEPLOYMENT += addFiles
}
```

The limitation of the above method is that all you can do is copy the file - you can't rename it, or take advantage of any of the other package file features. If you need to do anything other than copy the file then use the `DEPLOYMENT pkg_postrules`. The example below shows how to rename the file on installation:

```
symbian: {
addFiles.pkg_postrules =
"\"$(EPOCROOT)\\epoc32\\release\\$(PLATFORM)\\$(TARGET)\\myfile.dll\" -
\"!:\\sys\\bin\\myfile_installed.dll\""
DEPLOYMENT += addFiles
}
```

Supporting multiple languages

Creating a multilingual installation is more complicated. You need to declare which languages you support, and then provide a value for every other localisable string in the package, in the same order. Because the *default_deployment* package header, version, and dependencies are all defined for a single language, these need to be cleared. The post_rules can be left unchanged, because the executable is language neutral.

An example file is documented below, with explanation documented in-line:

```
# Clear all existing pkg dependencies, header, vendor id and language line (as these are
single language)
default_deployment.pkg_prerules =

# Define pkg languages  (in this case for English, French and Zulu, in that order)
languages = "&EN,FR,ZU(1024) "

# Define the package header. Note that the application name is translatable, and is
declared for each
   #language in the same order as the language line. As previously, the "" are escaped
using \ to satisfy the pro file syntax
   #You can use the same string for each if the value is the same in all languages
packageheader = "$${LITERAL_HASH}{\"MyApp-EN\", \"MyApp-FR\", \"MyApp-ZU\"},
(0x1000001F), 1, 2, 3, TYPE=SA"

#Define the language neutral global vendor name.
   #note that ";" is used to inject comments into the pkg file
vendorinfo = \
";The global vendor name (used to identify the package for package upgrades)" \
":\"GlobalVendorName\"" \

# The localised vendor name must have a value for each language, in the same order as
the language line
";The localised vendor name (a comma separated group of names in each language supported
by the package)" \
"%{\"Vendorname-EN\", \"Vendorname-FR\", \"Vendorname-ZU\"}"

#Define the dependencies. As above, you need strings for each translation of the
dependency and in the same order as
#the language line. In this case there is no translation, so the strings should be the
same.
dependencyinfo = \
"; Default HW/platform dependencies" \
"[0x101F7961],0,0,0,{\"S60ProductID\",\"S60ProductID\",\"S60ProductID\"}" \
"[0x102032BE],0,0,0,{\"S60ProductID\",\"S60ProductID\",\"S60ProductID\"}" \
"[0x102752AE],0,0,0,{\"S60ProductID\",\"S60ProductID\",\"S60ProductID\"}" \
"[0x1028315F],0,0,0,{\"S60ProductID\",\"S60ProductID\",\"S60ProductID\"}"

#Define the Qt component dependency. The string "Qt" can be used for all translations
  #Qt Mobility is defined similarly, but will use the Qt mobility version number.
qtdependency = "(0x2001E61C), $${QT_MAJOR_VERSION}, $${QT_MINOR_VERSION},
$${QT_PATCH_VERSION}, {\"Qt\",\"Qt\",\"Qt\"}"

# For smart installer, define name string in each language
DEPLOYMENT.installer_header = "$${LITERAL_HASH}{\"MyApp-Installer-EN\", \"MyApp-
Installer-FR\", \"MyApp-Installer-ZU\"},(0x12345678),1,2,3"

my_deployment.pkg_prerules = languages packageheader vendorinfo dependencyinfo
qtdependency
DEPLOYMENT += my_deployment
```

The above code will ensure that all the visible prompts will be display in the mobile devices current language. The application may also have files that are language specific:

- Resource files
- Translation files
- Mobile device shell icons

There are two main strategies that can be adopted for deploying these files. Either the application can deliver all the files to the device and determine which to use at runtime, or the installation can install only those files needed. The later approach is better, because it reduces the footprint of the application on the device memory.

The mechanisms for declaring language dependent files is given here: Language Dependent Files. Again there are a number of approaches:

- Install files automatically based on the phone's current language
- Install files based on user language selection

```
# Declare language dependent files installed based on the language of the target phone.
# The language codes/numbers used are for English (01), French (02), Zulu (98)
# Note, here we've deployed Symbian resource files, but they could be any files
languagedependentfiles = \
";English"\
"if supported_language=01"\
"    \"HelloWorld.r01\" -\"!: \ resource\apps\HelloWorld.r01\""\
"endif"\
";French"\
"if supported_language=02"\
"    \"HelloWorld.r02\" -\"!: \ resource\apps\HelloWorld.r02\""\
"endif"\
";Zulu"\
"if supported_language=98"\
"    \"HelloWorld.r98\" -\"!: \ resource\apps\HelloWorld.r98\""\
"endif"\

#Note, languagedependentfiles must be added to the DEPLOYMENT variable too!
```

Conditional installation, options lists, displaying a license during installation, etc

The installation can be extended using *pkg_postrules* and the raw package file syntax, in the same way as done in the preceding section. Links are provided to the appropriate syntax below - feel free to extend this article with a fuller description/example in Qt format

- Conditional installation
  - Options-List - Display a list of options to the user during installation and use the result to control the installation of optional components.
  - Condition blocks - Install files based on the properties of the target device, and the results of functions (e.g. presence of a particular file)
- Display licence information or other text or run applications during installation/uninstallation - Specify additional options during normal file installation using *pkg_postrules*. For example, the following *package file syntax* displays a file during installation in a dialog with a continue button.

  ```
  "text\textfile.txt"-"", FT, TC
  ```

- Embedded SIS file - Deploy shared components (e.g. shared library DLLs) in their own SIS file, embedded in the application SIS file

NOTE: Feel free to extend this article by providing individual sections for each of the above topics.

Frequently Asked Questions

What is the package file format?

The Package File Format is fully described in the product reference. An overview of each of the package file sections is given below:

- Languages Supported

  The set of languages (and optional dialects) supported by the installation file. These are expressed in terms of Language codes.

- Package-Header

  The name of the application/package (in each of the supported languages), a unique identifier of the package, major version, minor version, package build number and package options (including whether this package is a stand alone application, a patch to a stand alone application or one of the other options).

- Vendor

  The vendor/owner of the package. There are two vendor types: the first is a set of localised strings for the vendor name in each language supported by the file which is used in several installation dialogs, and the second is a global vendor name, used to identify the vendor for package updates.

- Package-Body

  This is the main body of the package file, which includes the files to be installed and any SIS files that are to be embedded within this SIS file. It can also include condition blocks that display different prompts or install different files based on the properties of the target device (e.g. existence of a file etc.) and options-lists which can display options to the user during installation.

- Dependency

  The dependency section allows you to specify the set of components (SIS files) that must be present on the device in order for this package to install and run successfully. It also allows you to specify the set of platforms or products that the application is able to be installed on (the application will install on any listed device or platform dependency).

- PKG Properties

  Package properties allow you to create a specific property associated with your SIS file. The meaning of the property is up to the person defining it - and can be read and used by other SIS files for conditional execution.

Examples of the correct package file format for each section are given in the above links, and in the PKG File Examples.

Can I use any other file format for deploying native C++ applications?

No.

Symbian platform security ensures that executable code can only be loaded from the */sys/bin* directory, and that code can only be copied into */sys/bin* by the software installer. The software installer is therefore the gate for all software loaded into the phone, and it understands sis and sisx files. It is not possible to simply copy files into arbitrary locations in the file system and run them.

Nor is it easy to create your own installer - as it would require manufacturer approval to get the required `AllFiles` capability.

What if I want to separately package a DLL?

In order to create a SIS file for a DLL project, declare a `DEPLOYMENT` variable in your `.pro` file.

Qt automatically creates the default package files for applications (projects with `TEMPLATE = app`). Other target types, including DLL projects, do not automatically create package files until a `DEPLOYMENT` variable is specified.

What are the default Qt installation packages?

Qt automatically creates application and installer package files based on the project file. By default:

- The application package file specifies a English installation (only) that contains all the application executable files, is dependent on the Qt and Qt Mobility libraries, and which can run on any S60 3rd Edition and later device.
- The installer package *copies* the application SIS file and *embeds* a SIS file for the *smart installer*. It is otherwise similar to the application package file in terms of language support, vendor name, and version numbers. The UID used for the wrapper package is 0xA000D7CE - this will need to be changed if the application is to be Symbian Signed.

The project file for a basic wizard generated Symbian project, and its associated packages are shown below (select links to

expand). The behaviour of each file is marked up with "NOTE" in the file listing.

Default application project file

Default application template package

Default installer package

What is the Nokia Smart Installer for Symbian?

The Nokia Smart Installer is a tool that can be bundled along with your application in a wrapper SIS file. On installation, the Smart Installer checks whether Qt is present (in the right versions) and if needed it is downloaded and installed prior to installing the application. This approach means that the application installation file can be quite small, and that Qt is only downloaded and installed when needed.

For more information see Nokia Smart Installer for Symbian

Related Information

- Standard Symbian packaging toolchain