

QDK Cookbook

Quick & Easy Recipes for Healthy Packaging

Table of Contents

QDK Cookbook.....	1
Preface.....	4
Intended Audience.....	4
Conventions.....	4
Faster build process during development.....	5
Problem.....	5
Solution.....	5
Discussion.....	5
Handle unknown configuration files.....	6
Problem.....	6
Solution.....	6
Discussion.....	6
External configuration file.....	7
Problem.....	7
Solution.....	7
Discussion.....	7
Configuration file created at installation.....	8
Problem.....	8
Solution.....	8
Discussion.....	8
.qdksave files are created at upgrade.....	9
Problem.....	9
Solution.....	9
Discussion.....	9
A disabled service is restarted at reboot	10
Problem.....	10
Solution.....	10
Discussion.....	10
Same options have to be repeated at every build.....	11
Problem.....	11
Solution.....	11
Discussion.....	11
Source code repository meta files are included in package.....	12
Problem.....	12
Solution.....	12
Discussion.....	12
QPKG depends on Optware packages.....	13
Problem.....	13
Solution.....	13
Discussion.....	13
QPKG depends on an unofficial Optware package.....	14
Problem.....	14
Solution.....	14
Discussion.....	14
Add symbolic link to file in QPKG directory when enabled.....	15
Problem.....	15
Solution.....	15
Discussion.....	15
Different extra architecture files shall be included.....	16

Problem.....	16
Solution.....	16
Discussion.....	16
Include a directory for run-time data.....	18
Problem.....	18
Solution.....	18
Discussion.....	18

Preface

When using QDK to build QPKG packages there are some common situations that can turn up over and over again. This document is a collection of straightforward recipes to solve these situations and be able to build new QPKG packages in a simple and efficient way.

Intended Audience

This document is about how to handle different situations when creating QPKG packages using QDK. As such, it assumes that the reader already has some previous knowledge about QDK and QPKG packages. This document also assumes that you have a basic knowledge of common shell commands and some familiarity with shell scripts.

Conventions

The following conventions are used in this document.

Italic is used to indicate files, directories, commands, and program names when they appear in the body of a paragraph.

Constant Width is used in examples to show the contents of files or the output from commands, and to indicate environment variables, keywords, function names, and other code snippets in the body of a paragraph.

Constant Bold is used in examples to show input that is typed literally by the user.

is the shell prompt.

[] surrounds optional elements in a description of program syntax. The brackets themselves should never be typed.

➔ is the code-continuation character that is inserted into code when a line shouldn't be broken, but we ran out of room on the page.

Faster build process during development

Problem

During development you might have to modify the *package_routines* file multiple times and it takes a long time to re-build the QPKG package after each change.

Solution

Extract the QPKG package to a new directory and make the modifications to the *package_routines* file in this directory. To test the changes you only have to run `sh qinstall.sh` to perform an installation of the package.

Discussion

Large packages could take quite a while to build because first the data shall be added to a compressed tar archive and then this tar archive and different meta data files are concatenated into a QPKG package. Doing this every time you have made some minor modification to the *package_routines* file could slow down the development considerably.

The packaging is an unnecessary step to only test the changes of the package specific functions, so by working directly with the installation script we can short-circuit that part of the process.

Don't forget to add the modifications to the main *package_routines* file when done, though.

Handle unknown configuration files

Problem

When you start using QPKG_CONFIG for a configuration file that already exists in previous versions of the package the file is not restored the first time you upgrade the package (a copy is saved with a *.qdkorig* suffix, though).

Solution

Add the configuration file using `add_qpkg_config` in the package specific initialization function.

```
pkg_init(){  
    add_qpkg_config myApp.conf 6d7fce9fee47aa94118b5b6e47267f03  
}
```

Discussion

The situation with unknown configuration files usually occur when a QPKG package is created for an application that was previously installed manually or when there have been previous QPKG packages without support for handling configuration files (either because of non-QDK created QPKG packages or using an older version of QDK).

The reason that unknown configuration files are not restored automatically is that they cannot be trusted to work with a new package. To handle this situation the package builder must acknowledge that the unknown configuration files can be trusted at an upgrade in the `pkg_init` function. The package builder can do this by adding the configuration file and the md5sum to the `SYS_QPKG_CONFIG_FILE` using the `add_qpkg_config` support function.

The `add_qpkg_config` function takes two arguments, the name of the configuration file and the md5sum value. The md5sum value is for the original file (i.e. the one included in the package). For configuration files created at installation time the md5sum should be set to 0 (also see Configuration file created at installation). The `add_qpkg_config` function checks that the configuration file isn't already added to `SYS_QPKG_CONFIG_FILE` before adding it, so it won't modify existing values.

External configuration file

Problem

The configuration file must be located outside of the QPKG directory.

Solution

Use QPKG_CONFIG with full path and then place the configuration file (with path) in the *config* directory.

Discussion

While it is possible to place the configuration file in the QPKG directory and then create a symbolic link to it from the external location it is usually simpler to add it at the external location already at installation, but only if the external location is on an actual HDD volume (e.g. /etc/config). If it is located on the RAM disk then the symbolic link solution is the only viable solution.

Configuration file created at installation

Problem

The configuration file is created at installation time.

Solution

Add the configuration file using `QPKG_CONFIG` and then run `qbuild` with the `--force-config` option to build the package.

Discussion

The `qbuild` application checks that the specified configuration files are included in the package and exit with an error if any file is missing. The `--force-config` option can be used to indicate that we are indeed aware that specified configuration files are missing, but that they are going to be created at installation time.

This gives the result that the configuration files are automatically given an md5sum of 0, so the upgrade still handles the files correctly and doesn't replace a locally modified file.

.qdksave files are created at upgrade

Problem

At an upgrade the installed version of a configuration file is replaced by the configuration file included in the QPKG and a copy of it is saved with a .qdksave suffix.

Solution

Modify the installed configuration file with any required changes and set the MD5 sum for the configuration file to the same as the value for the configuration file that is included in the QPKG,

```
set_qpkg_config CONFIG_FILE MD5SUM
```

Discussion

When the installed configuration file has been modified, but the new and original files are not identical it is possible that the local modifications are not valid any longer, so the installed file is replaced with the new file. A backup of the installed file is, however, saved with the name `file.qdksave` and a message is written to the system log.

This is the generic behaviour, but the package builder, who has better knowledge about what has been modified in the configuration file can change this behaviour in the `pkg_init` function. Because the configuration file in the package has changed it might be necessary to modify the installed configuration file, too. Using for example `$CMD_GREP` to find out if the modification has already been performed by the user might be a good start, next it is possible to use `$CMD_SED` to add the modification at any location in the file or `$CMD_ECHO` to append it to the end of the file.

By updating the MD5 sum for the configuration file the package builder acknowledges that the installed configuration is up to date and can be used with the new package. This change must be performed in the `pkg_init` function, before the configuration files are compared in the pre-install phase.

A disabled service is restarted at reboot

Problem

Although the QPKG is disabled in the web interface the service is started after a reboot.

Solution

Modify the status check in the init-script and add an exit when the QPKG is disabled.

Discussion

Usually, this is only a problem when converting an existing QPKG to QDK. The original code probably look something like this.

```
if [ `/sbin/getcfg myApp Enable -u -d FALSE -f /etc/config/qpkg.conf` = UNKNOWN ]; then
    /sbin/setcfg myApp Enable TRUE -f /etc/config/qpkg.conf
elif [ `/sbin/getcfg myApp Enable -u -d FALSE -f /etc/config/qpkg.conf` != TRUE ]; then
    echo "myApp is disabled."
fi
```

Note that the check for a disabled QPKG only printout that the QPKG is disabled. This is probably not the expected behaviour. Still, when fixing this by including an 'exit 1' call after the printout it is also necessary to make sure the check is only run when the script is started. The reason for this is that the QPKG is disabled **before** the init-script is called, so the script would exit before the service was stopped if the check was run also when the service is to be stopped.

The above code snippet includes an unnecessary check when using QDK, though. In QDK there are only two different states for a QPKG application; enabled or disabled. When creating a new QPKG based on QDK's template then an init-script with the correct check is included by default.

```
start)
    ENABLED=$(/sbin/getcfg $QPKG_NAME Enable -u -d FALSE -f $CONF)
    if [ "$ENABLED" != "TRUE" ]; then
        echo "$QPKG_NAME is disabled."
        exit 1
    fi
```

When converting an existing QPKG package it is usually better to replace the current check with the QDK version.

Same options have to be repeated at every build

Problem

The same options have to be repeated every time the package is built.

Solution

Specify the options under a section in the user configuration file and then run `qbuild` with `-s SECTION`.

Discussion

For example, if `qbuild` is run with `--build-dir=/share/QPKG --setup setup.sh --strict` to build the package then a section could be added to `~/.qdkrc`

```
[myApp]
QDK_BUILD_DIR=/share/QPKG
QDK_SETUP=setup.sh
QDK_STRICT=TRUE
```

Now, `qbuild` can be run with `-s myApp` and the specified settings are used by default.

Source code repository meta files are included in package

Problem

Source code repository meta files (e.g. `.svn` for Subversion) is included in the package.

Solution

Use the `--exclude` option to specify the files that shall not be included. For example, `--exclude .svn/` to exclude the `.svn` directories and content.

Discussion

When the tar archive with the data files is created the files are first moved from the different directories to a common directory. If the `--exclude` option has been used then the files are excluded if they match the specified pattern. It follows the same pattern matching rules as the `rsync` command.

QPKG depends on Optware packages

Problem

The QPKG package depends on Optware packages and you want to make the installation work without any manual intervention.

Solution

Add the Optware packages to the QPKG_REQUIRE setting using `OPT/<package name>`.

Discussion

When checking the requirements the generic installation script will attempt to install any missing (or with wrong version) Optware packages. For example, if the QPKG requires the sed and rsync packages from Optware you would specify `QPKG_REQUIRE="OPT/sed, OPT/rsync"` in the QPKG configuration file (*qpkg.cfg*). If a specific version is required it is possible to include a version comparison, `=`, `!=`, `<`, `>`, `<=`, or `>=`, and a version specification.

QPKG depends on an unofficial Optware package

Problem

You have built your own Optware package that is not available in the central repository, but your QPKG requires this package.

Solution

Add it to QPKG_REQUIRE as described in *QPKG depends on Optware packages* and include the package in the QPKG package using QDK_EXTRA_FILE.

Discussion

When the QPKG package is built and an Optware package is included in a QDK_EXTRA_FILE definition then an index file (*Packages.gz*) is automatically generated and attached to the QPKG package. At installation the index file is used to create a local repository for the included Optware packages and the search path that is used at installation/upgrade of Optware packages is temporarily modified to also include the local repository.

Add symbolic link to file in QPKG directory when enabled

Problem

A symbolic link to an application in the QPKG directory should be created when the QPKG is enabled.

Solution

Get the location of the QPKG directory at run-time from the qpkg.conf file.

```
CONF=/etc/config/qpkg.conf
```

```
QPKG_NAME=myApp
```

```
QPKG_DIR=$(/sbin/getcfg $QPKG_NAME Install_Path -d "" -f $CONF)
```

Now, it is easy to create a symbolic link to an application in the directory.

Discussion

At installation different QPKG related information is registered in /etc/config/qpkg.conf. The location of the QPKG directory is registered to the Install_Path setting. Other information that can be retrieved from this file is if the QPKG is enabled or disabled (Enable), the date when the QPKG was installed (Date), the version (Version), the location of the init-script (Shell), etc.

After retrieving the location of the QPKG directory it is usually a good idea to check the returned result before using it. For example,

```
QPKG_DIR=$(/sbin/getcfg $QPKG_NAME Install_Path -d "" -f $CONF)
```

```
if [ ! -d "$QPKG_DIR" ]; then
```

```
    echo "$QPKG_DIR: no such directory"
```

```
    exit 1
```

```
fi
```

```
# Create symbolic link to application in QPKG directory
```

```
[ -x $QPKG_DIR/bin/myApp ] && /bin/ln -sf $QPKG_DIR/bin/myApp /usr/bin/myApp
```

Different extra architecture files shall be included

Problem

Different architecture specific files shall be included in the QPKG package.

Solution

Use a pre-build script to insert QDK_EXTRA_FILE settings in the QPKG configuration file and then a post-build script to remove the setting.

Discussion

When the QPKG package is built it runs a defined pre-build script before the QPKG package is built and then a defined post-build script after the build is finished.

If we add the extra file in the pre-build script for the specific architecture and then remove it again in the post-build script it is only included in the correct architecture specific QPKG package.

The pre-build script could include something like this (the first argument to the script is the architecture; empty for the generic build)

```
#!/bin/sh
case "$1" in
arm-x09)
    if ! /bin/grep -q "^QDK_EXTRA_FILE=\"myx09.tar.gz\"" $QDK_QPKG_CONFIG; then
        /bin/echo 'QDK_EXTRA_FILE="myx09.tar.gz"' >> $QDK_QPKG_CONFIG
    fi
    ;;
arm-x19)
    if ! /bin/grep -q "^QDK_EXTRA_FILE=\"myx19.tar.gz\"" $QDK_QPKG_CONFIG; then
        /bin/echo 'QDK_EXTRA_FILE="myx19.tar.gz"' >> $QDK_QPKG_CONFIG
    fi
    ;;
x86)
    if ! /bin/grep -q "^QDK_EXTRA_FILE=\"myx86.tar.bz2\"" $QDK_QPKG_CONFIG; then
        /bin/echo 'QDK_EXTRA_FILE="myx86.tar.bz2"' >> $QDK_QPKG_CONFIG
    fi
    ;;
esac
return 0
```

To remove the added setting we add this to the post-build script (in this example it is assumed that nothing else has been added to the QPKG configuration file after the QDK_EXTRA_FILE setting, i.e. it is the last line in the file.)

```
#!/bin/sh
[ -n "$1" ] && /bin/sed -i '$d' $QDK_QPKG_CONFIG
return 0
```

Now, we can run qbuild with the new build scripts.

```
# qbuild -pre-build prebuild.sh -post-build postbuild.sh
```

Include a directory for run-time data

Problem

A directory for run-time data should be available in the QPKG directory, but an upgrade should not modify the content in any way.

Solution

Create the directory in `pkg_install`

```
$CMD_MKDIR -p $SYS_QPKG_DIR/cache
```

Discussion

QDK only cares about the files included in the package , so if a directory is created in a package specific function it is unknown to QDK and will not be affected by an upgrade.