QDK - QPKG Development Kit

Makes Simple Things Easy and Hard Things Possible

Table of Contents

Preface

QDK is used to build QPKG packages. A QPKG package is a self-extracting archive with application files and meta-data. A QPKG package makes it easy for *anyone* to install and remove packages. It also gives a package maintainer almost total control on how the package is installed on the NAS.

The major design goal of QDK is to make it easy for the package maintainer to create a QPKG package. QDK started out as a simple modification of the first official release of the QPKG SDK, but now supersedes it. It includes many new features like architecture check at installation, support for digital signatures, different compression algorithms, a comprehensive option to check that other required QPKG packages or Optware packages are installed (or that conflicting packages are not installed), automatic installation of required Optware packages, and a powerful build script.

QDK is distributed under the GPL making it completely open and available for anyone to use.

Intended Audience

This document is about creating QPKG packages. As such, it assumes that the reader already has some previous knowledge about QPKG packages and at least know how to install and enable a QPKG package using the web interface. This document assumes that you have a basic knowledge of common shell commands. It also helps if you have 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.

Installation of QDK

Before we can start using QDK we have to install it. It is distributed as a platform-independent QPKG package and can be installed using the normal web interface. When installed and enabled QDK is ready to be used.

At installation a directory structure is created at the default QPKG location, which depends on whether it is installed on a RAID volume or a single drive. The exact location is available in the `QDK_PATH` variable in the system-wide configuration file, */etc/config/qdk.conf*, but it is usually */share/MD0_DATA/.qpkg/QDK* on a RAID system and */share/HDA_DATA/.qpkg/QDK* on a single drive system.

```
bin/
    qbuild
scripts/
    qinstall.sh
template/
    arm-x09/
    arm-x19/
    config/
    icons/
    shared/init.sh
    x86/
    x86_64/
    package_routines
    qpkg.cfg
qdk
```

qdk is the init script used to enable and disable QDK. This script is run automatically when QDK is enabled or disabled in the web interface. It creates and removes a symbolic link to the *qbuild* application in */usr/bin*.

qbuild is the build script.

qinstall.sh is the part of the installation script that includes the generic installation functions common to all QPKG packages. This file is never edited by the user. All user defined actions are added to a separate file, *package_routines*.

The *template* directory include files for a new build environment that can be created using *qbuild*. By default it includes common subdirectories where the content should be placed and templates for the package specific installation functions, *package_routines*, the QPKG configuration file, *qpkg.cfg*, and an init-script. The init-script, *init.sh*, is renamed to the same name as the package when the new build environment is created.

There are also four hidden files, *.qpkg_icon.gif*, *.qpkg_icon_80.gif*, *.qpkg_icon_gray.gif*, and *.uninstall.sh*. The three first files are the default icons used for QDK in the web interface and the last file is the script run to perform the required actions to clean up after QDK when it is removed using the web interface. This file is created automatically for all QDK based packages at installation time.

QPKG Configuration File

We begin with a description of the QPKG configuration file, which is required when building a QPKG package. The QPKG configuration file (by default a file named *qpkg.cfg*) defines common variables for the QPKG package. Most of the variables are only used at installation, but some of them are also used when the QPKG package is built.

This is an example of a configuration file,

```
QPKG_NAME="QDK"
QPKG_VER="2.0"
QPKG_AUTHOR="micke"
QPKG_LICENSE="GPLv2+"
QPKG_SUMMARY="QDK (QPKG Development Kit) is used to create QPKG packages."

QPKG_SERVICE_PROGRAM="qdk"
QPKG_RC_NUM="101"
```

The first three variables, `QPKG_NAME`, `QPKG_VER`, and `QPKG_AUTHOR`, must always be included.

`QPKG_NAME`

Name of the software being packaged. Usually, the name used for a package would be identical to the software being packaged. No white-space is allowed in the name or QDK will only use the first part of the name (up to the first space). The name must be 20 characters or less.

`QPKG_VER`

Version of the software being packaged. This should be as close as possible to the format of the original software's version. No white-space is allowed in the version. If it does, QDK will only use the first part of the version. It is recommended to only use alphanumeric characters, periods, and dashes. The value must be 10 characters or less.

`QPKG_AUTHOR`

Author or maintainer of the package (not necessarily the author of the software that is being packaged, though.)

The following variables are optional, but most QPKG packages would at least include `QPKG_RC_NUM` and `QPKG_SERVICE_PROGRAM`.

`QPKG_RC_NUM`

Preferred number in start/stop sequence. Note that the value is currently ignored by the QNAP NAS after a reboot at which point the actual number is assigned depending on the order of the applications in */etc/config/qpkg.conf* starting with 100 for the first application.

`QPKG_REQUIRE`

This field is used to make sure that any packages required for the current package to operate are installed. The format is a package name followed by an optional version comparison, =, !=, <, >, <=, or >=, and a version specification. It is possible to use | to separate requirements where only one of them has to be true. The package name is either another QPKG or it could be an Optware package which is indicated by adding an `OPT/` prefix to the package name. Optware packages are installed automatically if missing (or have incorrect version). If more than one requirement shall be included they must be separated by comma. If the specified requirement is not fulfilled then the installation is terminated with an error message to the system log.

Example, `QPKG_REQUIRE="Python >= 2.7, Optware | opkg, OPT/openssh"`

`QPKG_CONFLICT`

Logical complement to the `QPKG_REQUIRE` field. The `QPKG_CONFLICT` specifies what packages cannot

be installed if the current package is to operate properly. This field has the same format as the QPKG_REQUIRE field, namely, a package name optionally followed by a version comparison. It is not possible to use | to separate statements (neither is it logical to use it for QPKG_CONFLICT.)

Example, QPKG_CONFLICT="SSOTS"

QPKG_LICENSE

License for the packaged application. For example, GPLv2+, GPLv3+, BSD, or Commercial.

QPKG_SUMMARY

One-line description of the packaged application.

QPKG_CONFIG

Can be added to specify a configuration file that shall receive special handling when the package is upgraded. The value can be be a path relative to $SYS_QPKG_DIR or a full path.

When the installed configuration file has never been modified then it is replaced with the new file. When the installed file has been modified and the new and original files are identical it is assumed that the installed file is still valid and it is left unchanged. When the installed 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.

The last scenario is when the configuration file was not installed by a package with support for keeping track of configuration files, so it is not possible to determine whether the current file is OK or not. In this case the installed file is replaced with the one from the package (which is known to work), a backup of the installed file is saved with the name file.*qdkorig*, and a message is written to the system log.

Configuration files that are automatically generated at installation time can also be specified using QPKG_CONFIG, but in that case it is necessary to use the *--force-config* option or set QDK_FORCE_CONFIG to TRUE or the build fails because of missing configuration files.

QPKG_SERVICE_PROGRAM

Init-script used to control the start and stop of the installed software. This script shall support start, stop, and restart commands. Any other command is optional. It is not necessary to include an init-script, but it would seriously limit the use of most QPKG packages to not include one.

QPKG_SERVICE_PORT

Port number used by the installed application's service program.

QPKG_SERVICE_PIDFILE

Location where the PID of a running service shall be stored.

QPKG_WEBUI

Relative path to installed application's web interface (the specified path is relative the configured location of web server data; usually */share/Web* or */share/Qweb*.) The specified path must start with a '/'. The displayed link can only be accessed when the QPKG is enabled. A default value of '/' is set automatically at installation if QPKG_WEB_PORT has been given a value and QPKG_WEBUI is empty.

QPKG_WEB_PORT

Port number for the web interface. If empty and QPKG_WEBUI is defined then the default is to use the same port number as the web server (usually port 80).

The QPKG_ROOTFS and QPKG_SERVICE_PROGRAM_CHROOT variables are only applicable to TS-x09. If QPKG_ROOTFS is defined then QPKG_SERVICE_PROGRAM_CHROOT must also be defined.

QPKG_ROOTFS

Location of the chroot environment. If the value is empty then a default location of */mnt/HDA_ROOT/rootfs_2_3_6* is used.

`QPKG_SERVICE_PROGRAM_CHROOT`

Init-script that controls the start and stop of the installed software when running in the chroot environment. This script shall support start, stop, and restart commands. Any other command is optional. No default.

The following QDK specific variables can also be included in the configuration file to be used when the QPKG package is built. They are described in the *QDK Variables* chapter.

`QDK_DATA_DIR_ICONS, QDK_DATA_DIR_X09, QDK_DATA_DIR_X19, QDK_DATA_DIR_X86, QDK_DATA_DIR_X86_64, QDK_DATA_DIR_SHARED, QDK_DATA_DIR_CONFIG, QDK_DATA_FILE, QDK_EXTRA_FILE`

The *qbuild* application performs a simple sanity check of the QPKG configuration file when the build process is started. If the file is found to be corrupt then the build process exits. Also, *qbuild* will refuse to build a QPKG package unless `QPKG_AUTHOR`, `QPKG_NAME`, and `QPKG_VER` have been assigned values. If `QPKG_SERVICE_PROGRAM` is undefined then a warning will be issued, but the build will continue. If `QPKG_NAME` is found to be more than 20 characters or `QPKG_VER` more than 10 characters then the values are truncated with a warning. Any space in the name or version also results in a warning and truncation (only the string up to the first space is used).

Installation Script

The installation script is divided into two parts, one with generic installation functions common to all QPKG packages and one with package specific functions. Only the file with package specific functions must be created when building a QPKG package using QDK.

Generic Installation Script

The generic script, by default a file named *qinstall.sh*, performs the installation or upgrade of an already installed package in three steps, pre-install, install, and post-install.

In the pre-install phase, the base directory for the QPKG installation is located and system variables are assigned valid values, configuration files are handled, and the current status of the QPKG (enabled or disabled) is stored to be able to restore it later. The specified service program is stopped and then any package specific pre-install code is run.

In the install phase the data package is extracted in the QPKG directory, configuration files restored, and then any package specific install code is run.

In the post-install phase any QPKG icons are copied to the correct location, symbolic links for the service program are created, the QPKG is registered in */etc/config/qpkg.conf*, and then any package specific post-install code is run.

To replace the generic installation script with a different script the `QDK_INSTALL_SCRIPT` variable can be assigned the location of the script. The generic installation script includes several different system definitions that can be used in the package specific script (use them as if they were read-only with the only exception being `SYS_QPKG_SERVICE_ENABLED`, which could be changed if required).

`SYS_EXTRACT_DIR`

Path to directory with extracted files from QPKG packages. The value is assigned at run-time.

`SYS_HOSTNAME`

Host name for system the QPKG is installed on.

`SYS_CONFIG_DIR`

Path to directory where configuration files are stored. Default is */etc/config*

`SYS_INIT_DIR`

Path to directory where init-scripts are stored. Default is */etc/init.d*

`SYS_STARTUP_DIR`

Path to directory with symbolic links to init-scripts that shall be run at system startup. Default is */etc/rcS.d*

`SYS_SHUTDOWN_DIR`

Path to directory with symbolic links to init-scripts that shall be run at system shutdown. Default is */etc/rcK.d*

`SYS_RSS_IMG_DIR`

Path to directory with icons for the web interface. Default is */home/httpd/RSS/images*

`SYS_QPKG_BASE`

Base location. Always assigned the location of the volume with the Public share, for example, */share/MD0_DATA*.

`SYS_QPKG_INSTALL_PATH`

Base location of QPKG installed packages. Same as `$SYS_QPKG_BASE/.qpkg`

SYS_QPKG_DIR

Location of installed software. Same as $SYS_QPKG_INSTALL_PATH/$QPKG_NAME

SYS_QPKG_DATA_FILE_GZIP

Location of gzip compressed tar package with the data files. Default *./data.tar.gz*

SYS_QPKG_DATA_FILE_BZIP2

Location of bzip2 compressed tar package with the data files. Default *./data.tar.bz2*

SYS_QPKG_DATA_FILE_7ZIP

Location of 7-zip compressed tar package with the data files. Default *./data.tar.7z*

SYS_QPKG_DATA_FILE

Location of the tar package with the data files. Assigned the value of either SYS_QPKG_DATA_FILE_GZIP, SYS_QPKG_DATA_FILE_BZIP2, or SYS_QPKG_DATA_FILE_7ZIP at runtime depending on what type of data archive that is included in the QPKG package.

SYS_QPKG_DATA_MD5SUM_FILE

Location of the optional file with the md5sum values. Default *./md5sum*

SYS_QPKG_DATA_CONFIG_FILE

Location of the optional file with configuration files. Default *./conf.tar.gz*

SYS_QPKG_CONFIG_FILE

Location of system-wide configuration file for all installed QPKG packages. Default value is $SYS_CONFIG_DIR/*qpkg.conf. T*he following field names can be used when accessing the configuration file.

SYS_QPKG_CONF_FIELD_QPKGFILE, SYS_QPKG_CONF_FIELD_NAME, SYS_QPKG_CONF_FIELD_VERSION, SYS_QPKG_CONF_FIELD_ENABLE, SYS_QPKG_CONF_FIELD_DATE, SYS_QPKG_CONF_FIELD_SHELL, SYS_QPKG_CONF_FIELD_INSTALL_PATH, SYS_QPKG_CONF_FIELD_CONFIG_PATH, SYS_QPKG_CONF_FIELD_WEBUI, SYS_QPKG_CONF_FIELD_WEBPORT, SYS_QPKG_CONF_FIELD_SERVICEPORT, SYS_QPKG_CONF_FIELD_SERVICE_PIDFILE, and SYS_QPKG_CONF_FIELD_AUTHOR

SYS_QPKG_SERVICE_ENABLED

Used to determine if the QPKG should be enabled or disabled after the installation/upgrade is finished. At an installation the value is always set to FALSE, while at an upgrade the current status of the QPKG is assigned to this variable before any of the package specific functions are executed. If set to TRUE then the service program is also started at the end of the installation/upgrade (if it fails to start then the status is set to disabled).

SYS_PUBLIC_SHARE

Name of public system share.

SYS_PUBLIC_PATH

Location of public system share.

SYS_DOWNLOAD_SHARE

Name of system share for downloads.

SYS_DOWNLOAD_PATH

Location of system share for downloads.

SYS_MULTIMEDIA_SHARE

Name of system share for multimedia content.

`SYS_MULTIMEDIA_PATH`

Location of system share for multimedia content.

`SYS_RECORDINGS_SHARE`

Name of system share for recordings.

`SYS_RECORDINGS_PATH`

Location of system share for recordings.

`SYS_USB_SHARE`

Name of system share for USB content.

`SYS_USB_PATH`

Location of system share for USB content.

`SYS_WEB_SHARE`

Name of system share for web content.

`SYS_WEB_PATH`

Location of system share for web content.

Pre-defined command definitions.

`CMD_AWK, CMD_CAT, CMD_CHMOD, CMD_CHOWN, CMD_CP, CMD_CUT, CMD_DATE, CMD_ECHO, CMD_EXPR, CMD_FIND, CMD_GETCFG, CMD_GREP, CMD_GZIP, CMD_HOSTNAME, CMD_LN, CMD_LOG_TOOL, CMD_MD5SUM, CMD_MKDIR, CMD_MV, CMD_PKG_TOOL, CMD_RM, CMD_RMDIR, CMD_SED, CMD_SETCFG, CMD_SLEEP, CMD_SORT, CMD_SYNC, CMD_TAR, CMD_TOUCH, CMD_WGET, CMD_WLOG, CMD_XARGS,` and `CMD_7Z`

`CMD_PKG_TOOL` is assigned the path to either Optware's *ipkg* tool or opkg's *opkg* tool at runtime (if found).

Package Specific Installation Functions

The package specific functions (by default included in a file named *package_routines*) include any extra actions that shall be performed at installation. All the variables defined in the QPKG configuration file and in the generic installation script can be accessed in the package specific functions.

The following support functions can be used in the package specific functions.

`log "`*`MSG`*`"`

Outputs MSG to terminal (stdout) and system log.

`warn_log "`*`MSG`*`"`

Outputs MSG to terminal (stderr) and system log.

`err_log "`*`MSG`*`"`

Outputs MSG to terminal (stderr) and system log. Also alerts the web interface about the failure and terminates the installation/upgrade.

The error message always includes "`$QPKG_NAME $QPKG_VER` installation failed." followed by the message given to `err_log`. For example with a QPKG named myApp, version 0.1:

```
err_log "Data file not found."
```

would result in this message

```
myApp 0.1 installation failed. Data file not found.
```

`get_share_path "SHARE NAME" VARIABLE`

Assign location of given share to variable. For example,

`get_share_path openssh OPENSSH_PATH`

would assign the location on the HDD for */share/openssh* to a variable named `OPENSSH_PATH`.

`add_qpkg_config "CONFIGURATION FILE" MD5SUM`

Add configuration file with given 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. The 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.

This function could be used to make existing configuration files that were not previously specified with `QPKG_CONFIG` to be handled as if they actually were specified. See Creating a Simple QPKG Package for an example.

`set_qpkg_config "CONFIGURATION FILE" MD5SUM`

Update existing configuration file with given md5sum value.

`extract_data ARCHIVE [DIRECTORY]`

Extract specified tar archive to given directory. If the directory is not specified then the default is to use `$SYS_QPKG_DIR`. Can be used to extract any extra data archives that have been included in the QPKG package.

If a version check shall be added in one of the package specific functions then the following support function could be used to compare the versions. They all take two version strings as input and return 0 if the test is successful, otherwise they return 1.

`is_equal, is_unequal, is_less_or_equal, is_less, is_greater,`

`is_greater_or_equal`

The *package_routines* file has the following content by default.

```
######################################################################
# List of available definitions (it's not necessary to uncomment them)
######################################################################
###### Command definitions #####
#CMD_AWK="/bin/awk"
:
#SYS_WEB_PATH=""
#
######################################################################
# All package specific functions shall call 'err_log MSG' if an error
# is detected that shall terminate the installation.
######################################################################
#
######################################################################
# Define any package specific operations that shall be performed when
# the package is removed.
######################################################################
#PKG_PRE_REMOVE="{
#}"
#
#PKG_MAIN_REMOVE="{
#}"
#
#PKG_POST_REMOVE="{
#}"
#
######################################################################
# Define any package specific initialization that shall be performed
# before the package is installed.
```

```
#########################################################################
#pkg_init(){
#}
#
#########################################################################
# Define any package specific requirement checks that shall be
# performed before the package is installed.
#########################################################################
#pkg_check_requirement(){
#}
#
#########################################################################
# Define any package specific operations that shall be performed when
# the package is installed.
#########################################################################
#pkg_pre_install(){
#}
#
#pkg_install(){
#}
#
#pkg_post_install(){
#}
```

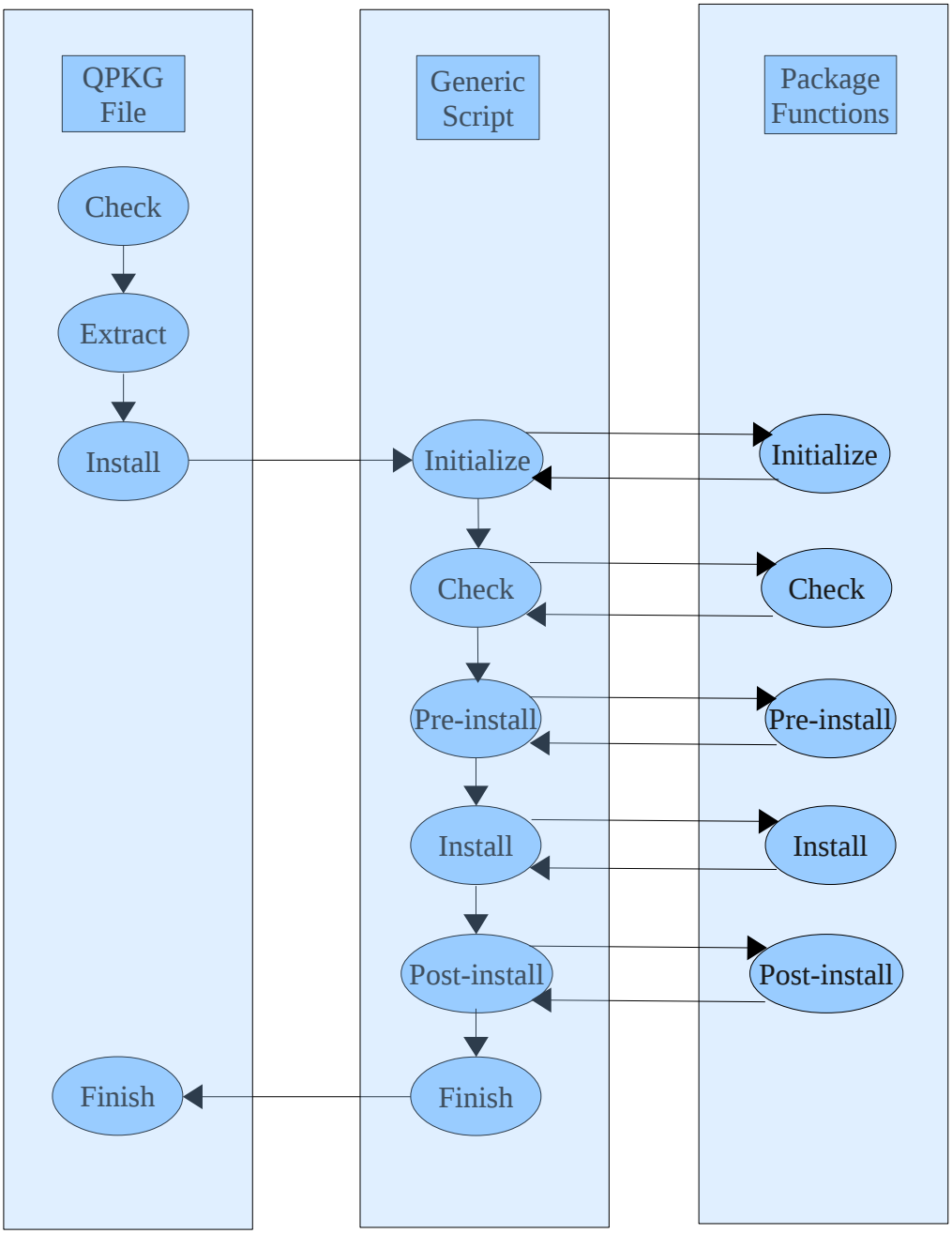
The package specific functions shall only return when the function has been successful or any detected error can be ignored, because the generic installation script ignores any returned values from the package specific functions. If an error is detected that should terminate the installation or upgrade then `err_log` shall be called with an error message.

`PKG_PRE_REMOVE, PKG_MAIN_REMOVE,` and `PKG_POST_REMOVE` are pseudo-functions that are included in the generated uninstall script. They can include almost the same code as a normal function, but variables defined in the functions must be escaped to be included in the uninstall script. For example, `$VAR` must be included as `\$VAR` or the variable is replaced with its value already when the uninstall script is created (which most likely would result in an empty value). Also, any command substitutions, `$(command)` (or the older format `` `command` ``) must be escaped if the command should be executed at runtime instead of when the uninstall script is created.

Order of Execution

Installation/Upgrade

When a QPKG package is installed or upgraded the following actions are run in a linear order.



First a check is performed to make sure that the mount point, */mnt/HDA_ROOT*, where the initial installation files should be extracted exists (this should always be true on a QNAP device). If it is a platform specific QPKG package then an architecture check is also performed to make sure the QPKG is installed on the correct platform. If both checks are successful then a temporary installation directory, */mnt/HDA_ROOT/update_pkg/tmp*, is created and the first level of QPKG packages are extracted to this directory – that is, the tar archive with the data files (default is *data.tar.gz*, but other compression algorithms are possible which would result in a different file suffix), the generic and package specific installation scripts (*qinstall.sh* and *package_routines)*, the QPKG configuration file (*qpkg.cfg)*, any extra data packages that

have been specified using QDK_EXTRA_FILE, and the optional files with MD5 checksums and configuration files. Next, the installation is started by running the generic installation script, *qinstall,sh*.

The generic installation script first locates the included data archive and assigns the file name to SYS_QPKG_DATA_FILE. System variables are initialized with name and path to the default system shares and the system definitions SYS_QPKG_BASE, SYS_QPKG_INSTALL_PATH, and SYS_QPKG_DIR are assigned valid values. If Optware packages are included in the QPKG package then a temporary local repository is assigned to the package tool's search path, so that the included files are available for a possible installation/upgrade in the requirement check for Optware packages. At an upgrade the list of installed files is sorted and renamed, so a new list can be created and later compared with the old list. This is followed by performing any package specific initialization.

Next it checks that any specified requirements are fulfilled. If any required Optware package is missing then the latest version of the package is installed automatically either from the remote repository or from a package included in the QPKG package (using QDK_EXTRA_FILE). After the check for any QPKG_REQUIRE or QPKG_CONFLICT definitions in the QPKG configuration file, the package specific requirement checks are performed (if defined). If the requirements are OK then the pre-install phase is started.

In the pre-install phase the QPKG directory specified in SYS_QPKG_DIR is created if it is a new installation and configuration files are handled according to what was specified when the package was built. At an upgrade the current status of the QPKG (enabled or disabled) is stored to be able to restore it later. The service program is stopped if it exists (usually not at a new installation) and then any package specific pre-install actions are performed.

The install phase starts with extracting the data from the data archive, $SYS_QPKG_DATA_FILE, to the QPKG directory, extracting any (optional) included full path configuration files to their correct location, and restore any previously stored configuration files, followed by any package specific install actions.

In the post-install phase any obsolete files from the previous installation are removed, any included QPKG icons are copied to */home/httpd/RSS/images/*, symbolic links for the service program are created in */etc/init.d*, */etc/rcS.d*, and */etc/rcK.d*, and the QPKG is registered in */etc/config/qpkg.conf*. This is followed by any package specific post-install actions.

Finally, the script that shall be run when the QPKG is removed from the system is generated (including any package specific actions), if the QPKG was enabled before the upgrade then at this point the service program is also started, and a success message is written to the system log and also reported back to the web interface.

After a new installation the default is for the QPKG to be disabled. If we would like to force the QPKG to be enabled from the beginning then it is possible to set SYS_QPKG_SERVICE_ENABLED to TRUE in one of the package specific functions, but it would be better to allow the user to decide when it should be enabled. After an upgrade the current status will be restored after the upgrade is finished. That is, if the QPKG was enabled before the upgrade then it is enabled after the upgrade and the service program is started (if it fails to start then the status is set to disabled, though.) This behaviour can be stopped by setting SYS_QPKG_SERVICE_ENABLED to FALSE in one of the package specific functions. If $SYS_QPKG_SERVICE_ENABLED is FALSE then the QPKG is always disabled after the upgrade.

QDK Variables

To configure the build process it is possible to assign values to different QDK variables. All the QDK prefixed variables can be included in the system wide configuration file, */etc/config/qdk.conf*, or in the user's configuration file, *~/.qdkrc*. The variables can also be included in the QPKG configuration file or build scripts. Command line arguments override any variables in the user's configuration file, which in turn override any variables in the system wide configuration file. Variables specified in the QPKG configuration file or in a build script can also override previous specified (or default) values.

`QDK_VERSION`

QDK version.

`QDK_PATH`

Path to where QDK is installed.

`QDK_USER_CONFIG_FILE`

Location of the user's configuration file. Default value is *~/.qdkrc*.

`QDK_QPKG_CONFIG`

Location (or expected location) of the QPKG configuration file. Default value for this variable is a file named *qpkg.cfg* in the current directory.

`QDK_PACKAGE_ROUTINES`

Location (or expected location) of file with package specific functions. Default value for this variable is a file named *package_routines* in the current directory.

`QDK_SCRIPTS_DIR`

Location of directory with the script files that are used by *qbuild*. Default value is `$QDK_PATH`*/scripts*.

`QDK_TEMPLATE_DIR`

Location of the directory with the template files that should be used when creating a new build environment using the *--create-env* option. Currently, generic versions of *qpkg.cfg* and *package_routines* shall be located in the specified directory. Any other files and directories are optional. Default value is `$QDK_PATH`*/template*.

`QDK_INSTALL_SCRIPT`

Location of the generic installation script. That is, the script that is run when the QPKG is installed. Default value is `$QDK_SCRIPTS_DIR`*/qinstall.sh*.

`QDK_VERBOSE`

Indicates *qbuild's* level of verbosity. 0 is quiet mode, 1 is normal mode, 2 is verbose mode, 3 is debug mode, and 4 is an extra verbose debug mode. Default value is `1`.

`QDK_STRICT`

Indicates if *qbuild* is run in strict mode or not. Default value is `FALSE`. In strict mode all warnings are treated as errors.

`QDK_FORCE_CONFIG`

When set to `TRUE` then *qbuild* ignores that specified configuration files are missing from the package. Can be used when the configuration files are created at installation of the QPKG package. Default is `FALSE`.

`QDK_COMPRESS_METHOD`

Indicates what compression method shall be used to compress the included files. Valid options are `gzip`, `bzip2`, and `7zip`. Default value is `gzip`.

`QDK_COMPRESS_FILE`

Name of the compressed data archive that is included in the QPKG package. Depends on the selected compression method. With `gzip` compression the name is *data.tar.gz*, with `bzip2` it is *data.tar.bz2*, and with `7zip` it is *data.tar.7z*.

`QDK_CONTROL_FILE`

Name of the included archive with meta-files (the generic and package specific installation scripts, the QPKG configuration file, and the optional files with MD5 checksums and configuration files.) Default name is *control.tar*. Note that this name is only used internally by *qbuild*; it is not used at the installation.

`QDK_SETUP`

Location of the setup script.

`QDK_TEARDOWN`

Location of the teardown script.

`QDK_PRE_BUILD`

Location of the pre-build script.

`QDK_POST_BUILD`

Location of the post-build script.

`QDK_ROOT_DIR`

Location of the root directory with the subdirectories, files, and meta-data that shall be used for the QPKG package. Default value is current directory.

`QDK_BUILD_DIR`

Location where the built QPKG package shall be placed. Default value is `$QDK_ROOT_DIR/`*build*.

`QDK_BUILD_VERSION`

Version that shall be used for the built QPKG package. Default is to use the `QPKG_VER` value from the QPKG configuration file, but if `QDK_BUILD_VERSION` is defined then that value is used instead and the `QPKG_VER` value in the QPKG configuration file is updated with the specified version.

`QDK_BUILD_MODEL`

Model the built QPKG package can be installed on. Default is to allow installation on any model. The check for the model tag is only performed by the web installation; installing a QPKG package on the command line is not affected by this setting.

`QDK_BUILD_ARCH`

QPKG packages shall be built for the specified architectures. Supported values are arm-x09, arm-x19, x86, and x86_64. Multiple values must be comma-separated. No default value; by default *qbuild* determines what to build based on the available files and directories in `$QDK_ROOT_DIR`.

An architecture check is added automatically when an architecture specific package is built. If at installation of the QPKG package the check fails then the installation exits with a system log message, `"Wrong architecture: $QPKG_NAME $QPKG_VER is built for `*`ARCH`*"`.

`QDK_RSYNC_EXCLUDE`

Exclude patterns for *rsync* that are used to not include files matching the pattern in the data package specified in `QDK_COMPRESS_FILE`. The format is `QDK_RSYNC_EXCLUDE="`*--exclude=PATTERN*"`. It is possible to specify multiple patterns, `QDK_RSYNC_EXCLUDE="`*--exclude=PATTERN1 --exclude=PATTERN2 ...*"`.

`QDK_RSYNC_EXCLUDE_FROM`

Location of file with exclude patterns. Similar to `QDK_RSYNC_EXCLUDE`, but specifies a file with exclude patterns (one per line). The format is `QDK_RSYNC_EXCLUDE_FROM="`*--exclude-from=FILE*".

`QDK_SIGN`

If set to `TRUE` then a digital signature is added to the QPKG package at build time. The *gpg2* application must be installed and in the package builder's path or the location specified in `QDK_GPG_APP` or the build fails with an error.

`QDK_GPG_APP`

Path to gpg2 application. Default is to search for it in the user's path.

`QDK_GPG_NAME`

Identity of private key that shall be used for the digital signature.

`QDK_GPG_PUBKEYRING`

Path to public keyring that shall be used to verify a digital signature. Default is */etc/config/qpkg.gpg*.

`QDK_GPG_KEYPATH`

Path to directory with default keyrings to be used when adding signatures. Default is the `$GNUPGHOME` environment variable, but if the keyrings are located at as location where gpg2 doesn't expect them this variable can be used to specify the location.

`QDK_SIGNATURE`

Use specified type of digital signature. Currently, only `gpg` is supported.

`QDK_DATA_DIR_ICONS`

Location of directory with icons for the packaged software. Default location is a directory named *icons* in `$QDK_ROOT_DIR`. The value must be a full path or a path relative to `$QDK_ROOT_DIR`.

The icons shall be named `${QPKG_NAME}`*.gif*, `${QPKG_NAME}`*_80.gif*, and `${QPKG_NAME}`*_gray.gif*,

- `${QPKG_NAME}`*.gif* is the image displayed in the web interface when the QPKG is enabled. It should be a GIF image of 64x64 pixels.
- `${QPKG_NAME}`*_gray.gif* is the image displayed in the web interface when the QPKG is disabled. It should be a GIF image of 64x64 pixels. It is usually a greyscale version of the `${QPKG_NAME}`*.gif* image, but that is not a requirement.
- `${QPKG_NAME}`*_80.gif* is the image displayed in the pop-up dialog (with information about the QPKG and the buttons to enable, disable, and remove). It should be a GIF image of 80x80 pixels.

If no icons are included then the QPKG is given default icons at installation.

`QDK_DATA_DIR_X09`

Location of directory with files specific to arm-x09 packages. Default location is a directory named *arm-x09* in `$QDK_ROOT_DIR`. The value must be a full path or a path relative to `$QDK_ROOT_DIR`.

`QDK_DATA_DIR_X19`

Location of directory with files specific to arm-x19 packages. Default location is a directory named *arm-x19* in `$QDK_ROOT_DIR`. The value must be a full path or a path relative to `$QDK_ROOT_DIR`.

`QDK_DATA_DIR_X86`

Location of files specific to x86 packages. Default location is a directory named *x86* in `$QDK_ROOT_DIR`. The value must be a full path or a path relative to `$QDK_ROOT_DIR`.

`QDK_DATA_DIR_X86_64`

Location of directory with files specific to x86 (64-bit) packages. Default location is a directory named *x86_64* in `$QDK_ROOT_DIR`. The value must be a full path or a path relative to `$QDK_ROOT_DIR`.

`QDK_DATA_DIR_SHARED`

Location of directory with files common to all architectures. These files are included before the

architecture specific files when the package is created to allow any architecture specific files to replace shared files. Default location is a directory named *shared* in $QDK_ROOT_DIR. The value must be a full path or a path relative to $QDK_ROOT_DIR.

QDK_DATA_DIR_CONFIG

Location of directory with full path configuration files. Default location is a directory named *config* in $QDK_ROOT_DIR. The value must be a full path or a path relative to $QDK_ROOT_DIR.

The complete path under the file system root (/) shall be created in this directory. For example, if a configuration file shall be installed in */etc/config/myApp.conf* then $QDK_DATA_DIR_CONFIG should contain the subdirectory structure *etc/config/* in which the *myApp.conf* file is placed.

Note that configuration files that are located relative to the QPKG directory ($SYS_QPKG_DIR) can be placed in any of $QDK_DATA_DIR_X09, $QDK_DATA_DIR_X19, $QDK_DATA_DIR_X86, $QDK_DATA_DIR_X86_64, $QDK_DATA_DIR_SHARED, or $QDK_DATA_DIR_CONFIG. It is only external configuration files that must be placed in $QDK_DATA_DIR_CONFIG.

QDK_DATA_FILE

Name of local data package that shall be used when creating the QPKG package. If defined then $QDK_DATA_DIR_ICONS, $QDK_DATA_DIR_X09, $QDK_DATA_DIR_X19, $QDK_DATA_DIR_X86, $QDK_DATA_DIR_X86_64, and $QDK_DATA_DIR_SHARED are ignored and no other data files or icons are included in QPKG package. If QDK_EXTRA_FILE has been specified then the extra data packages are still included, though.

QDK_EXTRA_FILE

Name of extra data package that shall be included in the QPKG package. The value must be a full path or a path relative to $QDK_ROOT_DIR. Any included extra data package must be extracted by package specific functions. It is possible to add multiple QDK_EXTRA_FILE to specify more than one extra data package. If Optware packages are assigned to QDK_EXTRA_FILE variable then an index file (*Packages.gz*) for all included Optware packages is generated automatically and attached to the QPKG package. This file is used at installation to create a local repository with the included Optware packages and include it in a possible installation/upgrade of required Optware packages.

QDK_QPKG_FILE

Name of the last built QPKG package (stored in $QDK_BUILD_DIR). Can be accessed in the post-build script to perform any kind of actions on the package (for example zip the file and upload it to a remote location, although if more than one package is created it might make sense to wait with the upload until the teardown script is run after all packages have been built). $QDK_QPKG_FILE is reset before the pre-build script is run to allow the script to set a new value, before a default value is assigned.

Build Scripts

QDK supports four different types of build scripts, one that is called once before the actual build process is started, one that is called before each build, one that is called after each build, and finally, one that is called once after the build process is finished. If a script returns a non-zero value then the build process is interrupted and the *qbuild* application exits with an error message. The build scripts should use the `return` command to return the status and not the `exit` command. If `exit` is used then not only the build script exits, but the build process itself, too.

The script specified with *--setup* (or the `QDK_SETUP` variable) could be useful for any global changes before the build process is started. When the defined setup script is called is has access to all the QDK prefixed variables, but many of them will not be assigned any values, yet.

Also, when the script is run *qbuild* has not checked that the configuration file exists. In other words, it would for example be possible to generate a configuration file in the script and then, if necessary, update the `QDK_QPKG_CONFIG` to specify the new location. Similar to `QDK_QPKG_CONFIG` the `QDK_PACKAGE_ROUTINES` variable is also still unchecked at the time when the setup script is run, so this file could also be generated in the script or the location to a generic file could be specified.

If a QPKG configuration file has been created by the setup script or it has been verified that there is a valid QPKG configuration file at the location specified in `$QDK_QPKG_CONFIG` then the values in the configuration file can be modified by the setup script using one of *qbuild*'s support functions,

```
edit_qpkg_config FIELD VALUE [location of QPKG configuration file]
```

If the location of the file isn't specified then the default location specified in `$QDK_QPKG_CONFIG` is used.

The next script is the pre-build script, specified with the *--pre-build* option or by using the `QDK_PRE_BUILD` variable. This script could be useful to make any package specific changes before each build, for example architecture specific handling like building a binary for the current architecture or include a pre-packaged archive for the current architecture..

When the pre-build script is run the `$QDK_QPKG_CONFIG` and `$QDK_PACKAGE_ROUTINES` have been checked to make sure they exist. The `QDK_BUILD_ARCH` variable have been assigned values if not already assigned.

The pre-build script is called with two arguments, the first one is set to the architecture for which the QPKG file is about to be built, and the second argument is set to the location of the architecture specific files. For the generic build the arguments are empty.

This script has access to the same variables as the setup script and also all the QPKG prefixed variables from the QPKG configuration file. The QPKG prefixed variables can be changed using `edit_qpkg_config`.

Note that if the `QDK_DATA_FILE` variable has been assigned a value for an existing file then the steps later in the build process to create a data package are skipped and the file specified in `$QDK_DATA_FILE` is used instead (although renamed to one of *data.tar.gz*, *data.tar.bz2*, or *data.tar.7z* when included in the QPKG package; the name depends on the compression method used for the specified tar archive in `$QDK_DATA_FILE`).

If `QDK_BUILD_VERSION` was specified in the user configuration file, in the setup or pre-build scripts, or if *--build-version* option was specified on the command line then the QPKG configuration file is updated to set the `QPKG_VER` field to the specified version before the QPKG package is built.

After the QPKG package has been built for the selected architecture (or generic) the post-build script is called, specified with the *--post-build* option or by using the `QDK_POST_BUILD` definition. It has access to the same variables as the pre-build script and for any file operations on the built QPKG package, the file can be found in `$QDK_BUILD_DIR` with the name `$QDK_QPKG_FILE.`

The post-build script also receives the architecture and location of architecture specific files as arguments (although the location of the files might be of less use to this script).

After the post-build is finished the build continues for the next architecture or if nothing more to build the last script, teardown, is called.

The teardown script, specified with the *--teardown* option or by using the `QDK_TEARDOWN` definition, has access to the same variables as all the previous scripts, although many of them would be of limited use considering that this script performs the last few actions before the build process exits.

All the scripts can use the following support functions for error messages, warning messages, normal messages, verbose messages, and debug messages

```
err_msg MSG
warn_msg MSG
msg MSG
verbose_msg MSG
debug_msg MSG
```

Actual output depends on `$QDK_VERBOSE` and `$QDK_STRICT` settings. A call to `err_msg` will always result in a termination of the build process and if `$QDK_STRICT` is set to TRUE then `warn_msg` gives the same result.

User Configuration File

The configuration file can be used to add customized settings for the various QDK variables. Any entries in the file must be added to sections. Each section has a name, which is enclosed in square brackets, followed by variable definitions. A section can include different QDK prefixed variables. Configuration sections are often useful for specific often-used groups of options. It is possible to define these options in a section of the configuration file and then just specify the section as the argument to *qbuild*.

By default, the `DEFAULT` section is included and then any sections specified on the command line using *--section SECTION* (or *-s SECTION*). Several sections can be included by repeating the option.

For example, if `QDK_PRE_BUILD=`*pre_build.sh* is added to the `DEFAULT` section in *~/.qdkrc*, then we could add a script named *pre_build.sh* to any project and when *qbuild* is run it will automatically run this script as part of the pre-build actions. If the location of a specific project, in this example Python, is specified then it would be possible to run '*qbuild -s python*' anywhere and *qbuild* would still build the project at the correct location.

```
[DEFAULT]
QDK_PRE_BUILD=pre_build.sh

[python]
QDK_ROOT_DIR=/share/QPKG/Python
QDK_VERBOSE=0
```

It is possible to specify a different location for the user configuration file in the system-wide configuration file, */etc/config/qdk.conf*, using the `QDK_USER_CONFIG_FILE` variable. For example, if all QPKG development if performed using the admin user then it might be easier to place the configuration file on the actual HDD instead of */root/.qdkrc*, which would be lost at each reboot.

Invoking qbuild

As the previous sections have shown, we can specify most options for each QPKG package in the configuration file and many times this is enough. But additionally, the *qbuild* application provides some global control through powerful command-line options and also a possibility to use different configurations that have been set up in the user configuration file.

The *qbuild* application has the following options

```
usage: qbuild [--extract QPKG [DIR]] [--create-env NAME]
      [-s|--section SECTION] [--root ROOT_DIR] [--build-arch ARCH]
      [--build-version VERSION] [--build-model MODEL]
      [--build-dir BUILD_DIR] [--force-config] [--setup SCRIPT]
      [--teardown SCRIPT] [--pre-build SCRIPT] [--post-build SCRIPT]
      [--exclude PATTERN] [--exclude-from FILE] [--gzip|--bzip2|--7zip]
      [--sign] [--gpg-name ID] [--verify QPKG] [--add-sign QPKG]
      [--import-key KEY] [--remove-key ID] [--list-keys]
      [--query OPTION QPKG] [-v|--verbose] [-q|--quiet] [--strict]
      [-?|-h|--help] [--usage] [-V|--version]
```

Initialize a Build Environment

To create a template build environment in the current directory the *--create-env* option can be used. It creates a directory with the specified name and copies the template files to the directory (replacing some fields in the QPKG configuration file with default values.) If *--build-version* is also used then the specified version is included in the generated QPKG configuration file. A template init-script is created in the subdirectory named *shared.*

Control the Build

When building a QPKG package it is possible to change some global settings. To change the location of the files and meta-data that shall be used for the QPKG package the *--root* option can be used (default location is to use the files in the current directory, '.'). The version is by default set to the same as the value of the `QPKG_VER` variable in the QPKG configuration file, but it can also be specified on the command line using the *--build-version* option, which also updates `QPKG_VER` in the QPKG configuration file.

When *qbuild* is run it determines what architectures that QPKG packages shall be created for by looking at what data directories that are not empty. It is possible to override this and specify what architectures that shall be included by using the *--build-arch* option (supported values: arm-x09, arm-x19, x86, and x86_64). Only one architecture can be specified per option, but it is possible to repeat the option on the command line to add multiple architectures.

When configuration files have been specified using `QPKG_CONFIG` then *qbuild* checks that the specified files exists or exits with an error. When the specified configuration file is created at installation then the *--force-config* option can be used to ignore that the file is missing from the package itself and still handle the file correctly at an upgrade.

The result is by default placed in a directory named *build* relative to `$QDK_ROOT_DIR`, but the *--build-dir* option can be used to specify a different directory for the result. If a full path is not specified then the specified location is relative to `$QDK_ROOT_DIR`.

Using the *--build-model* option it is possible to add an extra check for a given model tag at installation. If the given tag doesn't match the model for the device the QPKG package is installed on then the web interface fails to install the package.

It is possible to specify the compression methods using *--gzip*, *--bzip2*, or *--7zip*. By default the files are compressed using `gzip`.

Trust but Verify

When a QPKG package is installed the installation script runs with super-user privileges and a malicious package could cause all kinds of problems. As long as the QPKG is downloaded from a trusted package builder this is probably not an issue, but better safe than sorry. If the QPKG is signed then it is possible to use the *--verify* option to verify that it is built by the package builder and that it hasn't been modified. For this to work the package builder's public key must have been imported in the public keyring using the *--import-key* option.

All imported keys can be shown using the *--list-keys* option and a key can be removed from the keyring using *--remove-key*.

Of course, to be able to verify a package it must have been signed when it was built; this is accomplished using either the *--sign* or the *--gpg-name* option at build time or an existing package can have a signature added using *--add-sign*. The *--add-sign* option can also be used when an already signed package should be re-signed with a different key.

Note that only QPKG packages built with QDK 2.0 or later can have a signature added using the *--add-sign* option.

Exclude Files

Sometimes there might be files that shall be excluded from the QPKG package and by using the *--exclude* option it is possible to exclude files matching the specified pattern. This option is passed on to rsync and follows the same rules as rsync's --exclude option. Only one exclude pattern per option, but we can repeat the option on the command line to add multiple patterns. The *--exclude-from* option is related to the *--exclude* option, but specifies a file that contains exclude patterns (one per line). This option is also passed on to rsync and follows the same rules as rsync's --exclude-from option.

Scripts

One of *qbuild's* most powerful features is the script support. It makes it possible to run scripts at certain phases in the build process. The scripts can run any available shell commands and have access to both QPKG and QDK defined variables. This was discussed in greater details in a previous chapter.

The *--setup* option can be used to run a specified script to setup the build environment. It is called once before the build process is initiated. To clean up after all builds are finished the *--teardown* option can be used to specify the script to run. Called once after all builds are completed.

To control the build of a QPKG package the *--pre-build* option can be used. The specified script is run before the build process is started and is called before each and every build. First argument contains the architecture (one of arm-x09, arm-x19, x86, and x86_64) and the second argument contains the location of the architecture specific code. For the generic build the arguments are empty. It is also possible to specify a script that shall be run after the build process is finished with the *--post-build* option. It is called after each and every build with the same kind of arguments as the pre-build script.

Status Information

The application normally outputs errors and progress information to the terminal, but by using the *-q* or *--quiet* option we can change this. In quiet mode nothing is written to standard output. Normally only error messages are displayed. If quiet mode has been specified as the default mode then the *-v* or *--verbose* option can be used to enable normal output. By repeating the *-v* option the verbosity is increased. The maximum is 3 (4 if quiet mode has been specified as the default mode.)

By default only errors terminate the ongoing build process, while warnings are written to the terminal, but still allows the build process to continue. If the *--strict* option is specified then all warnings are treated as errors.

Sections

The --*section* option can be used to add a section to the list of searched sections in the configuration file. A section is a named set of definitions as described in the chapter about the user configuration file. By default, the `DEFAULT` section will be searched and then any sections specified on the command line.

Extract QPKG Packages

To access the content of a QPKG package the --*extract* option can be used to extract the archive of files and meta-data to a specified directory (default is the current directory).

Query Packages

The --*query* option can be used to retrieve information from a QPKG package without extracting the files. The following query options are available

- `dump` dump settings from qpkg.cfg
- `info` summary of settings in qpkg.cfg
- `config` list configuration files
- `require` list required packages
- `conflict` list conflicting packages
- `funcs` output package specific functions

Help and Usage

The --*usage* option display a brief usage information, while the -*?*, -*h*, or --*help* displays a more comprehensive help message about the options.

The -*V* or --*version* option prints a single line containing the version number of QDK.

Creating a QPKG Package Using QDK

Building a QPKG package using QDK is quite easy, especially if the software is platform independent and no binary has to be built or if a tar archive with the binary files is already available.

Normally, QDK enters the process when the binary has already been built, but by using some of QDK's more advanced features it is possible to include the build of the binary as part of the steps performed when building a QPKG package.

To learn more about QDK we start by creating a simple platform-independent QPKG package. Next we create QPKG packages for different platforms using some of QDK's advanced features. Finally, we convert an existing QPKG package to QDK.

Creating a Simple QPKG Package

The first simple QPKG package we create is in fact, the QDK itself.

To create a new QPKG package from scratch we can use *qbuild* to create a template directory with the basic structure.

```
# qbuild --create-env QDK
```

The command creates a directory named QDK with the contents from the template directory

```
arm-x09/
arm-x19/
config/
icons/
shared/QDK.sh
x86/
x86_64/
package_routines
qpkg.cfg
```

The *arm-x09*, *arm-x19*, *x86*, and *x86_64* directories are only used when building platform specific QPKG packages, so we remove those directories. Neither are we including any configuration files, so the *config* directory is also removed. The template init-script, *QDK.sh*, will be replaced by the one from the QDK installation directory, so it is also removed.

Next, we populate the remaining directories, *shared* and *icons*, with content. In this case, all the files are available in the QDK installation directory (as described in the section about the installation of QDK), so it is only a matter copying them to the directories (this example is on a RAID volume)

```
# rmdir arm-x09 arm-x19 x86 x86_64 config
# rm shared/QDK.sh
# cp -pr /share/MD0_DATA/.qpkg/QDK/* shared
# cp -p /share/MD0_DATA/.qpkg/QDK/.qpkg_icon.gif icons/QDK.gif
# cp -p /share/MD0_DATA/.qpkg/QDK/.qpkg_icon_80.gif icons/QDK_80.gif
# cp -p /share/MD0_DATA/.qpkg/QDK/.qpkg_icon_gray.gif icons/QDK_gray.gif
```

The directory structure should now look like this

```
icons/
    QDK.gif
    QDK_80.gif
    QDK_gray.gif
shared/
    bin/
        qbuild
    scripts/
        qinstall.sh
    template/
        arm-x09/
```

```
        arm-x19/
        config/
        icons/
        shared/
             init.sh
        x86/
        x86_64/
        package_routines
        qpkg.cfg
    qdk
package_routines
qpkg.cfg
```

The QPKG configuration file, *qpgk.cfg*, was created with default settings (the author is by default set to the name of the user running *qbuild*.)

```
QPKG_NAME="QDK"
QPKG_VER="0.1"
QPKG_AUTHOR="micke"

QPKG_SERVICE_PROGRAM="QDK.sh"
QPKG_RC_NUM="101"
```

We update the file to include a valid setting for the service program, *qdk*, specify the automatically generated configuration file that should be restored at an upgrade, license, one-line description, and also a different version number. All unused settings are removed leaving us with the following content.

```
QPKG_NAME="QDK"
QPKG_VER="2.0"
QPKG_AUTHOR="micke"
QPKG_LICENSE="GPLv2+"
QPKG_SUMMARY="QDK (QPKG Development Kit) is used to create QPKG packages."

QPKG_SERVICE_PROGRAM="qdk"
QPKG_RC_NUM="181"
QPKG_CONFIG="/etc/config/qdk.conf"
```

When QDK is installed we want the installation script to create a default system-wide configuration file in */etc/config/qdk.conf*. We could include a default file in *shared* and used that file at installation, but generating it at installation provides a more flexible solution. It also gives us an opportunity to introduce the *package_routines* file.

At installation we have to create the system-wide configuration file and then at a future upgrade we have to modify the settings in the existing configuration file, so we add a definition with the path to the system-wide configuration file at the top and script code to modify the file to *pkg_install*.

```
QDK_CONF=/etc/config/qdk.conf

pkg_install(){
        if [ -f "$QDK_CONF" ]; then
                $CMD_SED -i "s!\(QDK_VERSION=\).*!\1$QPKG_VER!" $QDK_CONF
                $CMD_SED -i "s!\(QDK_PATH=\).*!\1$SYS_QPKG_DIR!" $QDK_CONF
        else
                $CMD_ECHO "QDK_VERSION=$QPKG_VER" > $QDK_CONF
                $CMD_ECHO "QDK_PATH=$SYS_QPKG_DIR" >> $QDK_CONF
        fi
        $CMD_CHMOD 644 $QDK_CONF
}
```

When the package is removed the system-wide configuration file should also be removed, so a main remove function is also included,

```
PKG_MAIN_REMOVE="{
        $CMD_RM -f $QDK_CONF
}"
```

The addition of `QPKG_CONFIG="/etc/config/qdk.conf"` to *qpkg.cfg* makes sure that any local modifications are not lost at an upgrade. When *qbuild* is run it automatically adds a static checksum value for the configuration file to the *md5sum* file to handle this, however, we must use the *--force-config* option to indicate that the missing configuration file should not be treated as an error. Since the QDK 1.0 version also included this configuration file, but at that time no support existed for specifying configuration files, the configuration file is going to be handled as an unknown configuration file at the first upgrade, which would result in it being replaced by a new file (and the current file would be saved in */etc/config/qdk.conf.qdkorig*. To avoid this we use the package specific initialization function to specify that we trust the current file and that it can be handled as if it was a known configuration file. We accomplish this by adding the file to */etc/config/qpkg.conf* and setting the md5sum value to 0. This makes the installation script handle the configuration files as an already known file and the current file is untouched.

```
pkg_init(){
        add_qpkg_config $QDK_CONF 0
}
```

With all files updated we can run *qbuild* to create the QPKG package

```
# qbuild --force-config
Creating archive with data files...
Creating archive with control files...
Creating QPKG package...
```

The *qbuild* application figures out by itself that only a platform-independent QPKG package shall be created and the resulting QPKG package is placed in a directory named *build*. To change the location of where to store the result the *--build-dir* option can be used,

```
# qbuild --force-config --build-dir /share/QPKG/
```

Creating Platform Specific QPKG Packages

Next we create platform specific QPKG packages. For this example we create QPKG packages of One-Touch-Run for x86 and ARM x19. It is a very simple application, but it can still be used to show some QDK features that could come in handy when building for multiple architectures.

First a brief description of the development setup used in this example. The NAS itself doesn't include any development tools (apart from QDK); instead there are two build servers in the local network that perform the builds, one named *hugin* running Debian on ARM and one named *munin* running Debian on x86. To make sure that they work with pristine source code files they only work with files from a source code repository. That is, when the QPKG package is about to be built on the NAS all source code must first be checked in to the source code repository. This is not shown in the example, though. On the build server there is a user named *qdk* that accepts commands from remote hosts using *ssh*, for example to build a project. The result can be fetched using *scp*.

Just like when we created a single QPKG package we start by creating a template directory with a basic structure.

```
# qbuild --create-env One-Touch-Run
```

We remove the unused directories (*icons* and *arm-x09*) and create a new directory, *source*, for the source code to the One-Touch-Run library. We place the file with the C code in this directory together with a simple Makefile.

otr.c

```
#include <stdlib.h>

extern int Get_Usb_Copy_Avail();

int Get_Usb_Copy_Avail()
{
```

```
    (void)system( "/etc/init.d/one_touch_run run-script" );
    return 0;
}
```

Makefile

```
otr.so : otr.o
	$(CC) -o $@ -shared $?
.c.o :
	$(CC) -fPIC -c $?
clean :
	$(RM) *.o *.so
```

The files in the *source* directory are added to the source code repository to make them available to the build servers.

To be able to enable and disable the One-Touch-Run feature we edit the template init-script in *shared.*

```
#!/bin/sh

SCRIPT_DIR=/share/OTR/
CONF=/etc/config/qpkg.conf
QPKG_NAME=One-Touch-Run

usage()
{
    echo "Usage: $0 start|stop|restart|run-script"
    exit 1
}

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

        if [ -f $OTR_LIB ]; then
            /sbin/daemon_mgr gpiod stop "/sbin/gpiod &"
            /sbin/daemon_mgr gpiod start "LD_PRELOAD=$OTR_LIB /sbin/gpiod &"
        else
            echo "$OTR_LIB: No such file"
            exit 1
        fi
        ;;

    stop)
        /sbin/daemon_mgr gpiod stop "/sbin/gpiod &"
        /sbin/daemon_mgr gpiod start "/sbin/gpiod &"
        ;;

    restart)
        $0 stop
        $0 start
        ;;

    run-script)
        ENABLED=$(/sbin/getcfg $QPKG_NAME Enable -u -d FALSE -f $CONF)
        if [ "$ENABLED" = "TRUE" ] && [ -d $SCRIPT_DIR ]; then
            for script in $(/usr/bin/find $SCRIPT_DIR -type f)
            do
              if [ -x $script ]; then
                $script
              fi
```

```
            done
        fi
        ;;

    *)
        usage
        ;;
esac
exit 0
```

The init-script has support for the three required commands, *start*, *stop*, *restart*, and also an extra command *run-script* that is called when the button is pushed to run the user-defined script.

To make adjustments during the build for the different platforms we use the script support in *qbuild*. First we add a configuration file for the user in *~/.qdkrc* and in the DEFAULT section we specify that if there is a script named *setup.sh* in the directory then it is run before the build process is initiated (the setup phase).

```
[DEFAULT]
QDK_SETUP=setup.sh
```

By placing it in the DEFAULT section we can reuse this for any other project without adding new sections to the configuration file every time a new project is created.

A different solution that is project specific is to add a section named OTR and then specify the scripts and architectures in the section.

```
[OTR]
QDK_SETUP=setup.sh
QDK_PRE_BUILD=build.sh
QDK_TEARDOWN=cleanup.sh
QDK_BUILD_ARCH="arm-x19,x86"
```

In this case it would be necessary to specify the section name when running *qbuild.*

```
# qbuild -s OTR
```

The *setup.sh* script is used to create pre-build and teardown scripts. It also handles key management to access the remote build servers (that perform the actual build of the One-Touch-Run libraries for the different platforms) and specifies the platforms, so we don't have to do that on the command line. The pre-build script, *build.sh*, is used to instruct the build servers to perform the build, fetch the library, and clean up on the build server, while the teardown script, *cleanup.sh*, is used to clean up after the QPKG packages have been built.

setup.sh

```
#!/bin/sh
QDK_BUILD_ARCH="arm-x19,x86"

# Initialize key management
eval $(/opt/bin/ssh-agent)
/opt/bin/ssh-add

QDK_PRE_BUILD=build.sh
QDK_TEARDOWN=cleanup.sh

/bin/cat >$QDK_PRE_BUILD <<EOF
#!/bin/sh
ARCH="\$1"
HOST=
case "\$ARCH" in
arm-x19)
        HOST=hugin
        ;;
x86)
        HOST=munin
        ;;
*)
```

```
        HOST=
        ;;
esac
if [ -n "\$HOST" ]; then
        /opt/bin/ssh qdk@\$HOST 'otr-build'
        /opt/bin/scp qdk@\$HOST:/opt/result/OTR/otr.so \$ARCH
        /opt/bin/ssh qdk@\$HOST 'otr-clean'
fi
EOF
/bin/cat >$QDK_TEARDOWN <<EOF
#!/bin/sh
/opt/bin/ssh-add -D
/opt/bin/ssh-agent -k
/bin/rm -f arm-x19/otr.so x86/otr.so
/bin/rm $QDK_PRE_BUILD
/bin/rm $QDK_TEARDOWN
EOF
```

The directory structure should now look like this

```
arm-x19/
shared/
       one_touch_run
source/
       Makefile
       otr.c
x86/
package_routines
qpkg.cfg
setup.sh
```

The QPKG configuration file has a simple content.

```
QPKG_NAME="One-Touch-Run"
QPKG_VER="0.5"
QPKG_AUTHOR="micke"
QPKG_LICENSE="GPLv3+"
QPKG_SUMMARY="Run user-defined script using One-Touch button."

QPKG_SERVICE_PROGRAM="one_touch_run"
QPKG_RC_NUM="110"
```

With all files updated we can run *qbuild* to create the QPKG package

```
# qbuild
Agent pid 1687
Enter passphrase for /home/micke/.ssh/id_rsa:
Identity added: /home/micke/.ssh/id_rsa (/home/micke/.ssh/id_rsa)
cc -fPIC -c otr.c
cc -o otr.so -shared otr.o
otr.so                                100% 6111     6.0KB/s   00:00
Creating archive with data files for arm-x19...
Creating archive with control files...
Creating QPKG package...
cc -fPIC -c otr.c
cc -o otr.so -shared otr.o
otr.so                                100% 5787     5.7KB/s   00:00
Creating archive with data files for x86...
Creating archive with control files...
Creating QPKG package...
All identities removed.
unset SSH_AUTH_SOCK;
unset SSH_AGENT_PID;
echo Agent pid 1687 killed;
```

When *qbuild* is run it checks the settings in the configuration file, *~/.qdkrc*, and finds that it should run a script named *setup.sh* if it exists. Running this script starts the ssh agent and the private key is added to the agent after asking for the passphrase. Using an agent for the key management means that we only have to enter the passphrase once instead of for every *ssh* operation. At the same time the two scripts, *build.sh* and *cleanup.sh*, are created in the background and *qbuild* is also instructed to build QPKG packages for both the x86 and arm-x19 architecture.

Before building a QPKG package for a specific architecture it first calls the defined pre-build script with the architecture as input argument. The pre-build script calls the correct build server to build the library, fetch the result, and cleans up on the build server. The result is then used for the QPKG package.

After QPKG packages have been built for all architectures the teardown script is called to terminate the key management and remove any temporary files (the built libraries and the pre-build and teardown scripts).

The QPKG packages can be found in the *build* directory.

Converting an Existing QPKG Package

In the last example we convert an existing QPKG package to use QDK. The QPKG package chosen for this task is Optware.

We create a new build environment and extracts the contents of the existing QPKG package to the *shared* directory.

```
# qbuild --create-env Optware
# qbuild --extract Optware_0.99.163_arm-x19.qpkg Optware/shared
qinstall.sh
Optware.tgz
qpkg.cfg
# cd Optware/shared
# tar xvf Optware.tgz
./
./Optware.sh
./.qpkg_icon_80.gif
./.qpkg_icon.gif
./.qpkg_icon_gray.gif
```

The directory structure should now look like this

```
arm-x09/
arm-x19/
config/
icons/
shared/
       .qpkg_icon_80.gif
       .qpkg_icon.gif
       .qpkg_icon_gray.gif
       Optware.sh
       Optware.tgz
       qinstall.sh
       qpkg.cfg
x86/
x86_64/
package_routines
qpkg.cfg
```

The *Optware.tgz* file is removed and the icons are renamed to *Optware.gif*, *Optware_80.gif*, and *Optware_gray.gif* and moved to the *icons* directory.

```
# rm Optware.tgz
# mv .qpkg_icon.gif ../icons/Optware.gif
# mv .qpkg_icon_80.gif ../icons/Optware_80.gif
# mv .qpkg_icon_gray.gif ../icons/Optware_gray.gif
```

We edit the template *qpkg.cfg* using the information we find in the *qpkg.cfg* file included in the package. Apparently, it is not possible to upgrade Optware, so any existing installation of Optware is considered a conflict that should terminate the upgrade.

qpkg.cfg

```
QPKG_NAME="Optware"
QPKG_VER="0.99.163"
QPKG_AUTHOR="QNAP Systems, Inc."

QPKG_SERVICE_PROGRAM="Optware.sh"
QPKG_RC_NUM="103"
QPKG_CONFLICT="Optware"
```

After the file has been updated we can remove the old *qpkg.cfg* file. We fix the path bug in the init-script, *Optware.sh*, and also replace the partially buggy handling to find the location of the Optware installation with a solution that checks for the location in /etc/config/qpkg.conf.

```
#!/bin/sh

CONF=/etc/config/qpkg.conf
QPKG_NAME="Optware"

_exit()
{
      /bin/echo -e "Error: $*"
      /bin/echo
      exit 1
}

case "$1" in
  start)
      ENABLED=$(/sbin/getcfg $QPKG_NAME Enable -u -d FALSE -f $CONF)
      if [ "$ENABLED" != "TRUE" ]; then
            _exit "$QPKG_NAME is disabled."
      fi
      OPTWARE_DIR=$(/sbin/getcfg $QPKG_NAME Install_Path -d "" -f $CONF)
      if [ -d "$OPTWARE_DIR" ]; then
            /bin/echo "Enable Optware/ipkg"
            /bin/rm -f /opt
            /bin/ln -s $OPTWARE_DIR /opt

            # adding Ipkg apps into system path ...
            /bin/grep "PATH=.*/opt/bin.*" /etc/profile >/dev/null 1>&2 || /bin/echo ➥
            "export PATH=\$PATH":/opt/bin:/opt/sbin >> /etc/profile
      else
            _exit "$OPTWARE_DIR: no such directory"
      fi
      ;;

  stop)
      /bin/echo "Disable Optware/ipkg"
      export PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/local/sbin

      /bin/sync
      /bin/sleep 1
      ;;

  restart)
      $0 stop
      $0 start
      ;;
  *)
      echo "Usage: $0 {start|stop|restart}"
      exit 1
esac
```

```
exit 0
```

Working with the *qinstall.sh* script from the Optware QPKG package we move any package specific code to *package_routines*. We also fix any bugs when moving the code, for example the bug that changes any existing NTP settings at installation (we change the default interval from every hour to once a day to put less stress on the time servers, though.) The main difference between the Optware packages for the supported platforms is the feed setting, so we add a pre-build script that updates the feed for each package. To make it even simpler we create a setup script that generates the pre-build script and also a teardown script to clean up after the build process. With the settings in *~/.qdkrc* from the previous example the setup script is run automatically when *qbuild* is run.

setup.sh

```
#!/bin/sh
QDK_BUILD_ARCH="arm-x09,arm-x19,x86"

QDK_PRE_BUILD=config.sh
QDK_TEARDOWN=cleanup.sh

/bin/cat >$QDK_PRE_BUILD <<EOF
#!/bin/sh
case "\$1" in
arm-x09)
        /bin/sed -i -e 's/TYPE=.*/TYPE="cs05q1armel"/' \
                -e 's/KMOD_TYPE=.*/KMOD_TYPE="tsx09"/' $QDK_PACKAGE_ROUTINES
        ;;
arm-x19)
        /bin/sed -i -e 's/TYPE=.*/TYPE="cs08q1armel"/' \
                -e 's/KMOD_TYPE=.*/KMOD_TYPE="tsx19"/' $QDK_PACKAGE_ROUTINES
        ;;
x86)
        /bin/sed -i -e 's/TYPE=.*/TYPE="ts509"/' \
                -e 's/KMOD_TYPE=.*/KMOD_TYPE=/' $QDK_PACKAGE_ROUTINES
        ;;
esac
EOF

/bin/cat >$QDK_TEARDOWN <<EOF
#!/bin/sh
/bin/rm $QDK_PRE_BUILD
/bin/rm $QDK_TEARDOWN
EOF
```

package_routines

```
TYPE=
KMOD_TYPE=

FEED=http://ipkg.nslu2-linux.org/feeds/optware/${TYPE}/cross/unstable
if [ -n "$KMOD_TYPE" ]; then
        CONF=/opt/etc/ipkg/${KMOD_TYPE}.conf
        KMOD_CONF=/opt/etc/ipkg/${KMOD_TYPE}-kmod.conf
        KMOD_FEED=http://ipkg.nslu2-linux.org/feeds/optware/➥
${KMOD_TYPE}/cross/unstable
else
        CONF=/opt/etc/ipkg/${TYPE}.conf
fi

pkg_install(){
        INSTALL_PATH="${SYS_QPKG_DIR}/tmp/install"
        $CMD_MKDIR -p $INSTALL_PATH
        cd $INSTALL_PATH

        $CMD_ECHO "Downloading Optware/Ipkg ..."
        ipk_name=$($CMD_WGET -qO- $FEED/Packages | $CMD_AWK '/^Filename: ipkg-opt/➥
```

```
{print $2}')
        $CMD_ECHO "$CMD_WGET ${FEED}/${ipk_name}"
        $CMD_WGET -t 5 -T 20 ${FEED}/${ipk_name} || err_log "Cannot download ➥
${FEED}/${ipk_name}."
        $CMD_ECHO "Sym-link /opt ..."
        $CMD_RM -f /opt
        $CMD_LN -s ${SYS_QPKG_DIR} /opt

        $CMD_ECHO "Installing Optware/Ipkg ..."
        $CMD_TAR -xOvzf ${ipk_name} ./data.tar.gz | $CMD_TAR -C / -xzvf - 2>/dev/null
        $CMD_MKDIR -p /opt/etc/ipkg && $CMD_ECHO "src/gz $TYPE $FEED" > $CONF
        [ -z "$KMOD_TYPE" ] || $CMD_ECHO "src $KMOD_TYPE $KMOD_FEED" > $KMOD_CONF
        $CMD_ECHO "Export /opt/bin to $PATH ..."
        export PATH=$PATH:/opt/bin:/opt/sbin

        $CMD_ECHO "Updating the latest ipkg feeds ..."
        /opt/bin/ipkg update

        $CMD_ECHO "Deleting installation directory ..."
        cd -
        $CMD_RM -r $INSTALL_PATH

        # enable NTP & specify NTP server and time interval
        NTP_ENABLED=$($CMD_GETCFG NTP "USE NTP Server" -d FALSE)
        if [ "$NTP_ENABLED" != "TRUE" ]; then
                $CMD_SETCFG NTP "USE NTP Server" TRUE
                $CMD_SETCFG NTP "NTP Server IP" "pool.ntp.org"
                $CMD_SETCFG NTP Interval 1
                $CMD_SETCFG NTP TimeUnit DAY
                /sbin/ntpdate "pool.ntp.org"
        fi
        $CMD_SETCFG Misc Configured TRUE
}
```

With all files updated we can run *qbuild* to create the QPKG packages

```
# qbuild
Creating archive with data files for arm-x09...
Creating archive with control files...
Creating QPKG package...
Creating archive with data files for arm-x19...
Creating archive with control files...
Creating QPKG package...
Creating archive with data files for x86...
Creating archive with control files...
Creating QPKG package...
```

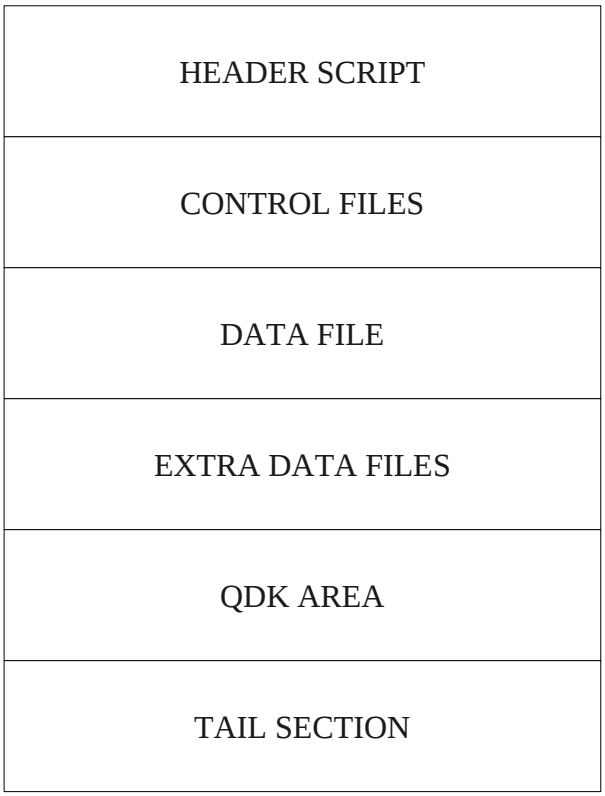
The QPKG packages can be found in the *build* directory.

Appendix A – QPKG Format

Simply described a QPKG file is a self-extracting archive that consists of a header, followed by the data, and last a tail section.

The header is a shell script that extracts the attached data and the tail section is 100 bytes of QNAP specific data used by the web interface at installation.

In QDK the data has been split into several parts and a QDK area has been added for QDK specific data. First a tar archive that contains the gzip compressed tar archive with the control files is attached, then the data archive (either a gzip, bzip2, or 7-zip compressed tar archive), which is followed by an optional tar archive with any extra data files (specified using `QDK_EXTRA_FILE`). If the QPKG file is signed then the signature is added to the QDK area.



Header Script

If the QPKG file is built for a specific architecture then QDK adds an architecture check to the header. The rest of the header script is code to create the directory for the files and to extract them. The header script is generated at build-time by the *qbuild* application.

Example of header script.

```
#!/bin/sh
wrong_arch(){
 local wrong_arch_msg="Wrong architecture: Optware 0.99.163 is built for arm-x19"
 echo "Installation Abort." && echo "$wrong_arch_msg"
 /sbin/log_tool -t2 -uSystem -p127.0.0.1 -mlocalhost -a "$wrong_arch_msg"
```

```
 echo -1 > /tmp/update_process && exit 1
}
arch_ok(){
 local cpu_arch=$(/bin/uname -m)
 [ $(/usr/bin/expr match "$cpu_arch" "armv5tel") -ne 0 ] || return 1
}
/bin/echo "Install QNAP package on TS-NAS..."
/bin/grep "/mnt/HDA_ROOT" /proc/mounts >/dev/null 2>&1 || exit 1
arch_ok || wrong_arch
_EXTRACT_DIR="/mnt/HDA_ROOT/update_pkg/tmp"
/bin/mkdir -p $_EXTRACT_DIR || exit 1
script_len=1032
/bin/dd if=${0} bs=$script_len skip=1 | /bin/tar -xO | /bin/tar -xzv -C $_EXTRACT_DIR ➥
|| exit 1
offset=$(/usr/bin/expr $script_len + 10240)
/bin/dd if=${0} bs=$offset skip=1 | /bin/cat | /bin/dd bs=1024 count=11 ➥
of=$_EXTRACT_DIR/data.tar.gz || exit 1
offset=$(/usr/bin/expr $offset + 10598)
( cd $_EXTRACT_DIR && /bin/sh qinstall.sh || echo "Installation Abort." )
/bin/rm -fr $_EXTRACT_DIR && exit 10
exit 1
```

Control Files

The control files include files as the generic and package specific installation scripts (*qinstall.sh* and *package_routines)*, the QPKG configuration file (*qpkg.cfg)*, and the optional files with MD5 checksums and configuration files. When the QPKG package is built these files are added to a gzip compressed tar archive, which is added to a uncompressed tar archive as a workaround to limitations in the available busybox tools. The header script extracts them to the installation directory.

Data File

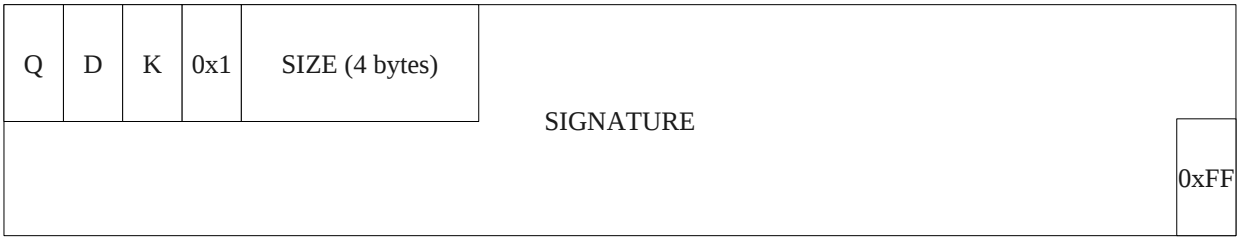
This is the compressed (gzip, bzip2, or 7-zip) tar archive with the actual data files which was created at build time with the files from the build root directory. The compressed tar archive is extracted to the installation directory and the installation script uncompress and extract the content to the QPKG directory. When the data archive is extracted to the installation directory some redundant data is included at the end of the data archive for performance reasons. The /bin/dd command on QNAP devices doesn't support different block sizes for the input and output, so to be able to extract the data at a reasonable speed the block size is set to 1024 bytes for both input and output. This gives the result that some extra data is added to the end of the archive (up to 1023 bytes depending on the original size of the archive). This doesn't matter since the tar archive includes an EOF marker and can be extracted without problems.

Extra Data Files

Any files specified in the QPKG configuration file using QDK_EXTRA_FILE are added to a tar archive at build time (no compression). The files in the included tar archive are extracted to the installation directory, where they are available for extraction by the package specific functions.

QDK Area

To be able to determine if a QDK area is included in the QPKG package the first three bytes of the area contains the text QDK which is followed by an optional amount of data blocks. Each block consists of a five bytes header followed by the actual data. The first byte in the header indicates what kind of data is stored in the block and the remaining four bytes contains the size of the data. The size shall be included in network byte order to make it platform-independent. Last in the QDK area is a one byte tag with the value 0xFF to indicate the end of the area. Currently, the only supported data type is 0x1 (digital signatures).



Tail Section

The tail section contains QNAP specific data that is used by the web interface at installation. It consists of 100 bytes, where the first 10 includes the model (`QDK_BUILD_MODEL`), followed by 40 unspecified reserved bytes (some of them are used for the checksum value), 10 bytes reserved for the firmware version (not supported in QDK), 20 bytes for the name (`QPKG_NAME`), 10 bytes for the version (`QPKG_VER`), and finally a 10 bytes flag area, which shall be set to "QNAPQPKG " for QPKG packages.