The WinLibre Packaging Policy

Introduction to Packages

A package is an archive that contains a software package. This includes installation scripts and binary files as well as meta-data such as the package description, version number and architecture.

This document is heavily based off the Debian Policy Manual (See References). The Debian package system has been tried and true for years now, providing one of the best and most reliable package systems available. The aim of WinLibre is to provide the equivalent of such package systems to Windows users and therefore it only makes sense to use the Debian package system as our basis.

Naming

Package filenames

Each package has a specific naming structure using the following format:

<package_name>_<version>_<architecture>.xxx

package_name is the name of the software. *version* is the fully formatted version string which will be covered in the next section. *architecture* is either "32bit" or "64bit".

.xxx – The file extension will be related to WinLibre. Some ideas are: .wlp (win libre package), .pkg, .libre

All packages are in the form of zip archives with a custom file extension. This allows users to extract the packages without natively being able to tell they are extractable.

Version Formatting

The version field is split up into three parts:

[epoch:]upstream_version[-winlibre_revision]

epoch – This is a single integer and is optional. This component is provided to allow mistakes in the version numbers. Normally it should be omitted in which case the epoch is assumed as 0 and no colons are allowed in *upstream_version*

upstream_version – This is the main part of the version number and is typically the original ("upstream") version from the original release. This section is mandatory and may need to be modified from the upstream version to be formatted properly. Allowed characters are [a-zA-Z0-9] and . + - : ~ and should start with a digit. If there is no *winlibre_revision* then no hyphens are allowed; if there is no *epoch* then no colons are allowed.

winlibre_revision – This optional component specifies the version of the WinLibre package based on the *upstream_version*. It may only contain [a-zA-Z0-9] and . + ~ and is compared in the same way *upstream_version* is. This is useful in the case where a specific WinLibre release is made. It is conventional to restart the *winlibre_revision* at 1 each time the *upstream_version* is increased. The package management system will break the version number apart at the last hyphen in the string (if there is one) to determine the *upstream_version* and *winlibre_revision*. If *winlibre_revision* is omitted, a value of 0 is assumed.

Version Comparison

When comparing two version numbers, first the *epoch* of each are compared, then the *upstream_version* if *epoch* is equal, and then the *winlibre_revision* if *upstream_version* is also equal. *Epoch* is compared numerically. The *upstream_version* and *winlibre_revision* parts are compared by the package management system using the following algorithm:

The strings are compared from left to right.

First the initial part of each string consisting entirely of non-digit characters is determined. These two parts (one of which may be empty) are compared lexically. If a difference is found it is returned. The lexical comparison is a comparison of ASCII values modified so that all the letters sort earlier than all the non-letters and so that a tilde sorts before anything, even the end of a part. For example, the following parts are in sorted order from the earliest to the latest: ~~, ~~a, ~, the empty part, a. (One common use of ~ is for upstream pre-releases. For example, 1.0~beta1~svn1245 sorts earlier than 1.0~beta1, which sorts earlier than 1.0.)

Then the initial part of the remainder of each string which consists entirely of digit characters is determined. The numerical values of these two parts are compared, and any difference found is returned as the result of the comparison. For these purposes an empty string (which can only occur at the end of one or both version strings being compared) counts as zero.

These two steps (comparing and removing initial non-digit strings and initial digit strings) are repeated until a difference is found or both strings are exhausted.

Note that the purpose of epochs is to allow us to leave behind mistakes in version numbering, and to cope with situations where the version numbering scheme changes. It is *not* intended to cope with version numbers containing strings of letters which the package management system cannot interpret (such as ALPHA or pre-), or with silly orderings (such as this example package whose versions went: 1.1, 1.2, 1.3, 1, 2.1, 2.2, 2 and so forth)

See the Debian Policy Manual for more information on versioning. WinLibre uses the same version formatting system.

FILES

Each package is required to contain an info.xml file. This file contains all the meta-data for a package as defined in Section 4 of this document.

Files to be installed may be included anywhere inside the archive. The recommended storage location for files is inside a directory instead of in the root of the archive. For simplicity's sake, binary files are recommended to be stored in a bin folder, configuration files in a config folder.

DESCRIPTION

Each package is required to contain an **info.xml** file containing the necessary meta-data to distinguish the package's needs and description. This consists of, title, description, dependencies and other information. The info.xml format is loosely based on the XPKG Meta-data format as well as Debian packages. The root element is <package> which contains all the other elements.

Name – This is a mandatory field stating the name of the binary package. Must consist of only lower case letters (a-z), digits (0-9), plus (+), minus (-), and periods (.) This must be at least two characters long and must start with an alphanumeric character.

Version – The version of the package, see Section IIB

Architecture – This is a mandatory field with a value of either “32bit”, “64bit”, or “any” stating the desktop architecture the package is intended for.

Short-Description – This is a mandatory field with a single line short description of the package.

Long-Description – The full description of the package.

Section – The category of software the package falls into. Should be one of the following: ADMIN, COMM, DEVEL, DOC, EDITORS, ELECTRONICS, EMBEDDED, GAMES, GNOME, GRAPHICS, HAMRADIO, INTERPRETERS, KDE, LIBS, LIBDEVEL, MAIL, MATH, MISC, NET, NEWS, OLDLIBS, OTHEROSFS, PERL, PYTHON, SCIENCE, SHELLS, SOUND, TEX, TEXT, UTILS, WEB, X11

Installed-size - The total amount of disk space required to install the named package. The disk space is represented in kilobytes as a simple decimal number.

Maintainer – The package maintainer's name and email address in <"first last" email@address.com> format.

Original-Maintainer – The original package maintainer's name and email address in <"first last" email@address.com> format.

Filename – The full path to the file on the server. The filename should be formatted in the style documented in Section IIA

Size – The total amount of disk space required to download the package. The disk space is represented in kilobytes as a simple decimal number.

SHA256 – The SHA256 of the archive. MD5 and SHA1 have recently discovered flaws and make them a less secure choice for package hashing.

Homepage - The URL of the web site for this package, preferably (when applicable) the site from which the original source can be obtained and any additional upstream documentation or information may be found. The content of this field is a simple URL without any surrounding characters such as <>.

RELATIONSHIPS

These relationships determine the availability of other packages when a user wishes to install a certain package. Some may conflict, some may require other packages to be installed. Here is a list of available relationships each package may have with other packages.

Replaces – This declares a list of packages that are no longer necessary as the current package replaces their functionality. See Dependency Formatting for formatting rules.

Provides – This is for virtual packages (See Debian Policy for Virtual Packages) which are not named after the actual package but provide all the functionality of the package. See Dependency Formatting for formatting rules.

Pre-Depends – This declares a list of packages that are required to be installed before the current package can be installed. See Dependency Formatting for formatting rules.

Depends - This declares an absolute dependency. A package will not be configured unless all of the packages listed in its Depends field have been correctly configured. The Depends field should be used if the depended-on package is required for the depending package to provide a significant amount of functionality. The Depends field should also be used if the postinstall, preremove or postremove scripts require the package to be present in order to run. See Dependency Formatting for formatting rules.

Recommends - This declares a strong, but not absolute, dependency. The Recommends field should list packages that would be found together with this package in all but unusual installations. See Dependency Formatting for formatting rules.

Suggests - This is used to declare that one package may be more useful with one or more others. Using this field tells the packaging system and the user that the listed packages are related to this one and can perhaps enhance its usefulness, but that installing this one without them is perfectly reasonable. See Dependency Formatting for formatting rules.

Conflicts – This declares a list of packages with which the current package should not be installed with on the same system. See Dependency Formatting for formatting rules.

The relations allowed are <<, <=, =, >= and >> for strictly earlier, earlier or equal, exactly equal, later or equal and strictly later, respectively. Version relations are allowed to be omitted where as the entire package will assume the relation, no matter which version. The package names listed may also include lists of alternative package names, separated by vertical bar (pipe) symbols |. In such a case, if any one of the alternative packages is installed, that part of the dependency is considered to be satisfied.

Here is an example info.xml file:

```xml
<package>
  <name>Example</name>
  <version>1.0-1</version>
  <architecture>32bit</architecture>
  <short-description>This is an example package</short-description>
  <long-description>This is my longer description. It includes\n
  newlines and other characters.</long-description>
  <section>devel</section>
  <installed-size>1024</installed-size>
  <maintainer>Chris Oliver <excid3@gmail.com></maintainer>
  <original-maintainer>Chris Oliver <excid3@gmail.com></original-maintainer>
  <replaces>
    <replace>example-package (>= 2.2.1)</replace>
  </replaces>
  <provides>
    <provide>some-package</provide>
  </provides>
  <pre-depends>
    <pre-depend>pre-package</pre-depend>
  </pre-depends>
  <depends>
    <depend>other-package | my-package</depend>
  </depends>
  <recommends>
    <recommend>another-package</recommend>
  </recommends>
  <suggests>
    <suggest>something</suggests>
  </suggests>
  <conflicts>
    <conflict>bad-package</conflicts>
  </conflicts>
  <filename>\pool\e\example_1.0-1_32bit.zip</filename>
  <size>100</size>
  <sha256>1a79a4d60de6718e8e5b326e338ae533</sha256>
  <homepage>http://excid3.com</homepage>
</package>
```

Scripting Files

Install and remove scripts are mandatory for each package. These are python scripts that will be executed by WinLibre during installation and removal of the package.

There are four optional scripts that a package may use as well: preinstall, postinstall, preremove, and postremove. These provide extra functionality if the package requires actions to be taken before the package before or after installation or removal of the package.

The following is a list of python scripts any package may use and their function:

preinstall.py – Executed before install.py.

install.py – main package installation script. This file is mandatory.

postinstall.py – Executed after install.py.

preremove.py – Executed before remove.py.

remove.py – Main package removal script. This file is mandatory.

postremove.py – Executed after remove.py.

These files MUST be stored in the root of the package archive. Scripts will be run as standard python script, so execution will begin as normal. No special functions or classes are required for the scripts to be executed properly.

Generally most packages will not need to make use of the pre/post scripts and they can therefore be omitted from the package. Some examples of packages which would use pre/post scripts would be ones that have a large install code base and would like to create desktop shortcuts post-installation as well as registering DLLs.

Install Scripting

Install scripts provide full functionality for common tasks during installation. This includes downloading of files, copying files from the package and performing registry changes. These functions are provided by the winblire.installer module.

The following is an example install script:

```
# WinLibre install script example

# Imports
import os
import os.path
from winlibre import installer

# Variables
pkg_name = ‘example’
uninstall_reg_location = ‘Software\Microsoft\Windows\’ \
                         ‘CurrentVersion\Uninstall\%s’ % pkg_name

# Setup the full directory path using the environment
# variable PROGRAMFILES
install_dir = os.path.join(os.getenv(‘PROGRAMFILES’),’example’)
uninstall = os.path.join (install_dir, ‘uinstall.exe’)

# Create the directory if it does not exist already
if not os.path.exists(install_dir):
    os.mkdir(install_dir)
```

```
# Copy example.exe from the package to the install directory
installer.copyFile(‘package://bin/example.exe’, install_dir)
installer.copyFile(‘package://bin/uninstall.exe’, install_dir)

# Add uninstall entry to Add/Remove Programs
installer.writeRegStr(‘HKLM’, uninstall_reg_location,
                         ‘DisplayName’, ‘Example Application’)
installer.writeRegStr(‘HKLM’, uninstall_reg_location,
                         ‘UninstallString’, uninstall)
```

So let’s break this down a bit. First off, you see that this script is just a standard python script. Normal entry point, exits when it falls off the end of the script. On to some of the WinLibre specific features

```
from winlibre import installer
```

This line imports the installer library provided by WinLibre for common tasks. Installer provides the following functions:

copyFile(source, destination**)** – Copies a file from the *source* location to its *destination*. s*ource* locations may start with “package://” to reference a file inside the archive, “http://” to reference a file remotely or “file://” to reference a file locally on the disk. *Destination* must be a location on the disk and should start with an environment variable such as PROGRAMFILES, which may be referenced by using os.getenv() from the os module. File locations should be concatenated using os.path.join().

executeFile(location**)** – Executes a file from given *location*

createShortcut(source, destination**)** – Creates a shortcut at *destination* linking to *source*.

createStartMenuEntry(source, destination**)** – Creates an entry in the Start menu just like createShortcut

writeRegStr(hive, path, key, value**)** – Writes *value* to the given registry location.

deleteRegStr(hive, path, key**)** – Deletes a registry key.

Other functions will be added as necessary. This is just a sample for now.

As you can see, WinLibre provides common functions that typical installers would use for installation. This makes I possible to create a complete installer using the provided functions as well as absolutely any tool provided by the python libraries that are currently installed. This provides flexibility for developers well versed in python as well as simple to use routines for those who have just begun.

OVERVIEW

That covers the package format so far. We have covered the package naming conventions, version system, file inclusion and install scripting.

This section will have a simple overview of each section.

REFERENCES

Debian Policy Manual

http://www.debian.org/doc/debian-policy/ch-controlfields.htm

XPKG Meta-data Format

http://xtreeme.org/xpkg/metadata/0.2/metadata.xhtml

Add/Remove Programs Registry Information

http://nsis.sourceforge.net/Add_uninstall_information_to_Add/Remove_Programs