

Maemo Diablo Packaging Applications Training Material

February 9, 2009

Contents

1	Packaging Applications	2
1.1	Creating Debian packages	2
1.2	Packaging Basics	2
1.3	Dependencies	3
1.4	Packaging infrastructure	3
1.5	Debian packages	5
1.6	Installation process	7
1.7	Package relationships	7
1.8	Package control file (aka Debian control file)	8
1.9	A maemo example control file	9
1.10	Creating your package	10
1.11	Adding debugging support for a package	17
1.12	Building the package for a device	17
1.13	Installing packages into an Internet Tablet	18

Chapter 1

Packaging Applications

1.1 Creating Debian packages

This chapter will introduce you to the Debian packaging system. We will restrict ourselves to cover the tools that are of immediate interest when creating new packages. Issues like building your own package repositories are not covered here, but there is ample information available with Internet search engines.

1.2 Packaging Basics

Most modern Linux distributions utilise some kind of packaging system. Two of the most popular ones are the Red Hat package Manager and Debian's dpkg-based system (although it's mainly popular because of a tool called *apt-get*).

All packaging systems strive to solve the same problems. Mainly, how to keep track which file belongs to which software package, so that a group of files may be upgraded in a coherent fashion, or removed all together. Keeping track of files necessitates some kind of local database and packaging systems are quite different in which technical solutions they select.

Before packaging systems existed on Linux, it was common to distribute source code as source tarballs (`make dist` makes it easy now, but back then it was a bit more difficult). This made installing software quite easy when you knew the tools that were used in building the software, but for a novice, it was sometimes painful. Removing the software was still quite hard, since targets like `uninstall` weren't common.

Slackware was the first Linux distribution to think about developing a common mechanism for software distribution, and they chose to distribute software as binary packages (this is what most distributions do now). You could of course get the source code for the software as well, but ordinary users weren't so much interested in source code, but wanted to get the software running. Computers were somewhat slower back then as well, so compiling software sometimes took long times (2.0 series kernel building took over 24 hours). Hence, the idea that one could build the binaries once, collect the relevant configuration and manual files together and then distribute them, was seen as an improvement.

Slackware used tarballs as the distribution mechanism. Each package tarball also contained a script that was executed on software install and this script could

then modify some system configuration files so that the software could run (set the runlevel related symbolic links for example).

Along came Red Hat, and they decided that the Slackware way was ugly and they could do better. They wrote their own packaging system (which also uses compressed binary archives, but not tar.gz). Red Hat's packaging system could relatively easily be used to rebuild the binaries if one so wanted or needed. The RPM system has a Berkeley DB-based local database in which the system keeps track of files that were installed from each package. The RPM system also provides some searching tools into the database as well as other features.

The Debian project started to think about package management at about the same time and implemented a packaging system that would also contain dependency information about what a package required to be installed beforehand before it could be installed safely. The reason why Debian didn't use RPMs was that RPM back then didn't support dependencies and wasn't available when Debian started thinking about these things. Nowadays both RPM and the Debian packaging system contain similar technical capabilities and it can be a matter of taste which one to choose when starting a new Linux distribution.

1.3 Dependencies

Soon after introducing the concept of software packages, it was evident that there needed to be some way to codify the rules within the packages which would guard against installing incompatible packages into the same system. The typical example was installing a mail server when there were multiple different choices for one (there have been alternatives to *sendmail* for many years). Suppose you wanted to install an alternative mail server and you already had *sendmail* installed. Since both of the services would access the same directory paths (mail spools), this would create chaos. They could not listen to the same TCP port either (not on the same IP address at least). This would cause problems to the administrator, so a system for describing package conflicts was developed. Soon after this, people noticed that applications started to use shared libraries and it didn't make sense to link libraries in static form into binaries, so a need for describing dependencies between installed packages arose.

In all modern Linux distributions it is the dependencies and their quality that determine how easy or hard it is for the busy system administrator to manage a Linux server. This applies to the desktop arena as well.

We will cover the syntax and operation of the Debian package dependencies and conflicts but other packaging systems have similar mechanisms.

1.4 Packaging infrastructure

The one thing that set apart the Debian packaging tools for a long time was a program called *apt-get*. It does not install software packages into the system, neither does it modify the local database. It is a program that reads package dependency and conflict information and tries to solve them in an intelligent way so that users may install or remove the packages that they want to.

One thing that makes apt-get special is its support for multiple ways of accessing repositories of packages (physical storage for packages in Debian). One of the most popular mechanism to use apt-get is the http-method(ftp being available as well). This means that when installing a package (which you don't have to download first), apt-get will first do some calculations and tell you which other packages (and their versions) need to be installed so that the original (and other) packages' dependencies won't be broken. After pressing ENTER, apt-get will download the necessary packages and invoke the low-level package management tool(dpkg) in the order necessary to complete the user's wishes.

One point needs to be re-iterated: writing correct dependencies (not too strict, but complete) is critical for tools like apt-get to work properly. Otherwise things will break and users will be unhappy. This is often the hardest thing to get right.

Some important program options and locations:

- `/etc/apt/sources.list` : a file listing the repositories and methods for apt-get.
- `/var/cache/apt/archives/` : apt-get downloads .deb-files here.
- `/var/lib/dpkg/` : location of the local package database (used with dpkg).
- `/var/lib/dpkg/info/` : control files for already installed packages.
- `apt-get install foo` : install binary package foo.
- `apt-get remove foo` : remove binary package foo.
- `apt-get source foo` : download source package for package foo.
- `apt-get build-dep foo` : download and install the packages required to build foo.
- `dpkg-source -x foo_version-revision.dsc` : extract the source package into a directory.
- `cd foo-version*; dpkg-buildpackage -rfakeroot -b` : build a binary package from a previously extracted directory.
- `dpkg -i foo_version-revision_arch.deb` : install the binary package file.
- `dpkg -l` : list installed packages.
- `dpkg -L foo` : list installed files of package foo.
- `dpkg -S /path/to/file` : find out which package "owns" the given file.
- `dpkg --info foo_version-revision_arch.deb` : show information stored in a package file.
- `dpkg-deb --contents X.deb` : list files inside a package file.

- `dpkg-deb --extract X.deb` : extract files from a package file (does not install).
- `apt-cache search keyword` : scan through the repository package lists while looking for the keyword and display package names which match.
- `apt-cache show packagename` : show information about package packagename (using the repository lists).

Please note that these commands and paths will be present on any real Debian system, but might be absent from special purpose target devices. An Internet Tablet for example might not contain all of these, and also some files have been removed to conserve storage space. The SDK environment contains them and you can even use `apt-get` to install new software from the `maemo.org` public repositories and get source code for most of the programs and libraries. This will require an connection to the Internet. You can sometimes even do an upgrade to a newer version of the SDK (you should check the SDK release notes and installation instructions before doing this).

1.5 Debian packages

Packages generally contain all the files necessary to implement a set of related commands and/or features.

There are two kinds of Debian packages:

- Binary packages (one **.deb** file): Contain executables, configuration files, man/info pages, copyright information and other documentation. This file is meant for end-user to install with `dpkg` on their system and contains compiled versions of software. Each deb-file is built for a specific architecture with specific compilation and build flags. This means that a .deb-file built for the i386-architecture will not run on ARM-architecture. The target architecture of the deb-file is given in the filename of the file (**sopwith_1.7.1-1_i386.deb** where the last part before .deb is the architecture). Some binary packages consist of files that are architecture independent (scripts or documentation) and their architecture is 'all' (**python-imaging+_1.1.4-3.1_all.deb** and **perl-doc_5.8.4-8_all.deb**)
- Source packages (at least one **.dsc** file and **.orig.tar.gz** file): These packages are used to distribute source code that can be used to create the binary packages. Most software is not Debian-specific, so you will normally find three files: The .dsc file that describes this source package, the source code of the original version (.tar.gz) and a compressed diff-file that contains the changes to be applied to the source code so that the software will adhere to the Debian policy. Sometimes these changes also include bug fixes. These changes are distributed as a .diff.gz-file (an unified difference file produced by the `diff`-tool).

Combined, these three files contain the source code, information and scripts necessary to build a binary package (or sometimes multiple binary packages from the same source package).

The package filename is structured as follows:

foo_versionNumber-maemoRevision_arch.deb

Package filename components:

- **foo**: name of package
- **versionNumber**: upstream software version (i.e., original non-Debian version)
- **maemoRevision**: `.diff.gz` has changed, while upstream version not.
- **arch**: name of target architecture (for binary package files)

The `maemoRevision` field might change when the control file or installable configuration files change. It will also sometimes change when a quick bug fix is done but the upstream maintainer hasn't yet released a version that includes this bug fix.

Some examples from one release of the SDK follow:

- `libdb4.2_4.2.52-18osso_armel.deb`:
 - `libdb4.2`: Binary package name. Package versions with incompatible APIs will normally have their package name contain a number.
 - `4.2.52-18osso`: Upstream version is `4.2.52`, with local modifications for maemo at version `18osso`.
 - `armel`: Binary package for the `armel` architecture.
- `libdb4.2_4.2.52-18osso_i386.deb`: Binary package for the `i386` architecture ("X86").
- `db4.2_4.2.52-18osso.diff.gz`: Differences between upstream and maemo version of the package. One part of the source package.
- `db4.2_4.2.52-18osso.dsc`: Debian Source Control file (defines a source package file). One part of the source package.
- `db4.2_4.2.52.orig.tar.gz`: Source code tarball of the upstream version. One part of the source package.

One noteworthy thing above is that source package names don't always correspond to binary package names. This is especially true with large source package which will produce multiple binary packages, or will package documentation separately.

The binary packages that can be built from one source package are listed using the `Binary` field of the Debian source control file:

```
Format: 1.0
Source: db4.2
Version: 4.2.52-18osso
Binary: libdb4.2+, db4.2-doc, libdb4.2-dev, libdb4.2+-dev, libdb4.2
Maintainer: Debian Berkeley DB Maintainers
<pkg-db-devel@lists.aliases.debian.org>
Architecture: any
Standards-Version: 3.6.1
Build-Depends: procps [!hurd-i386]
Uploaders: Clint Adams <schizo@debian.org>,
Matthew Wilcox <willy@debian.org>,
Andreas Barth <aba@not.so.arh.org>
Files:
cbc77517c9278cdb47613ce8cb55779f 4073147 db4.2_4.2.52.orig.tar.gz
4926da646ea05246767da25aac139aef 80499 db4.2_4.2.52-18osso.diff.gz
```

If a package is significantly different from the upstream version, it won't always have a `.diff.gz` file.

1.6 Installation process

When a package is installed by `dpkg`, the installation goes through various stages in the order specified below (from the `dpkg` man-page):

- Extract control files of the new package.
- If this is a package upgrade, execute the `'prerm'`-script of old version.
- Execute `'preinst'`-script if one is provided in the package.
- Backup old files (if upgrading) and unpack new files into their locations.
- If upgrading, execute `'postrm'` of the old package.
- Configure the installed package (`--configure`).

1.7 Package relationships

Each package contains a `control`-file which contains all the dependency, conflict and feature information. These are collectively called package relationships.

Debian supports the following kinds of relationships:

- **Pre-depends:** Similar to `'Depends'` (below) but meant for enforcing ordering of installation. **DO NOT USE.**
- **Depends:** Package will not work without specified version (or newer) of another package. Note that `Depends` rules are used after the package has been unpacked just before it's about to be configured. In some cases this might leave the package installed but non-configured (rare).
- **Recommends:** Recommended package will be useful to most users of this package.
- **Suggests:** Suggested package might include additional functionality and be useful to some users.
- **Enhances:** Similar to suggests but works in reverse direction with respect package order.
- **Conflicts:** This package will not operate correctly if the conflicting package is already installed.
- **Replaces:** The referenced package will be replaced by this package (by overwriting of original package files). When used together with `Conflicts`, a package replacing a conflicting package will cause the conflicting package to be removed first.

- **Provides:** Used to note that some function is provided by this package. Not used in maemo.
Syntax for the package specifications when declaring the relationships is as follows:
- Multiple package names are separated by a comma if the restriction must be against all of them (boolean AND).
- If one or more packages can satisfy the restriction, separate their names with the '|' (pipe) character (boolean OR).
- If version number of some package is significant, you can specify the version restriction by using «, <=, =, >= and » and a version number.

Some examples:

Depends: foo (=1.2.0)

Depends on package foo's exact version 1.2.0 (no other installed version will do)

Conflicts: foo

Conflicts with all versions of package foo

Depends: foo (>=1.2.3) | foobars, foozonkle

Package installation requires that package foo's version 1.2.3 or higher is installed, or package foobars is installed. Package foozonkle must be installed as well (irrespective of the previous restriction) but any version of it will do.

1.8 Package control file (aka Debian control file)

```
Package: packagename
Priority: optional (to aid intelligent tools wrt desktop installation)
Section: devel (which part of the ftp archives on Debian this package lives in)
Installed-Size: 45 (in KiB) to aid intelligent inst tools (filled in automatically)
Maintainer: First Lastname
Architecture: i386 (filled in automatically)
Version: 1.3-16 (-16 = Debian revision, filled in automatically)
Depends: libc6 (>= 2.1) (filled automatically when using shlibs:Depends-macro)
Description: The classic greeting, and a good example
Long description starts always with a space
Empty lines are not permitted (they terminate the long description)
So, there is a mechanism to allow this:
.
That was an "empty line"
```

An example Debian control file

For full syntax and canonical explanation, please see the Debian Policy Manual, Section 5.

Depends lists the packages which need to be installed for this package to install successfully (explained above).

Debian is split into three main sections: **main** (free software), **non-free** (not really free according to Debian policy), **contrib** (free software that depends on non-free).

However, in maemo, only one section has been defined so far: `user`. This has been further split into application categories which should be used when possible so that automatic localisation can be done in the Application manager.

The subsections as of this moment are:

- `user/accessories`: Accessories
- `user/communication`: Communication
- `user/games`: Games
- `user/multimedia`: Multimedia
- `user/office`: Office
- `user/other`: Other
- `user/programming`: Programming
- `user/support`: Support
- `user/themes`: Themes
- `user/tools`: Tools

If you cannot find a subsection that suits you, you can create a new one of your choosing, but it will not be automatically localised.

If you do not use the `user` section, the Application manager will not show your package as installable.

Each package is assigned a priority by the distribution maintainer (normally) to signify the relative importance with respect to proper functioning of the installed system. These control intelligent installation tools like `apt-get`. However, in maemo, only the priority `optional` should be used for your packages.

1.9 A maemo example control file

The Application manager supports special control directives as well as the normal Debian ones. This is how packages are provided with icons that the user will see on installation. These special directives start all with the prefix `XB-Maemo-`.

As example control file from the maemo.org HOWTOs looks like this:

```
Source: myapplication
Section: user/other
Priority: optional
Maintainer: Your Name <your.name@example.org>
Build-Depends: debhelper (>= 5)
Standards-Version: 3.7.2

Package: myapplication
Architecture: any
Depends: libhildon1 (>= 1.0.11)
Description: A simple test application
 A very simple application with a short description.
 Which spans multiple lines actually.
XB-Maemo-Icon-26:
iVBORw0KGgoAAAANSUUhEUGAAABoAAAAaCAYAAACpSkzOAAAABmJLR0QA/wD/AP+g
vaeTAAACXBtWXMMAAsTAAAEwEAMPwYAAAB3RJTUUH1gURDQoYya0JlWAAAU9J
REFUSMftl1KA0EUhb/NZ1/ggnHQxsJUxt5CUucVJCCkDfgyKdIGG5/A0s5HEBtJ
EdDAQGBgmw0YJmMzgXXYZa5CtNkDW9zZw5z7c+ZCgwb/Ai3i9sV1/Bq8RIs4LRK1
gJDsKvJyNXmJMuYTsMoY1zpgzoaABdYArQNPZQ1kfyGU7SpqVwxzAMwABWhgpIwp
4vWBB+AUWAi3ypjnfEXtPU4bLKx9vErTeCeIRSYF+fTn1j5dp2myE9EiU+DSi3wX
ymeqRQAmZ3EcA5E/fg06BULT8zh0crwXoJdrXRA2Lgps2y2odAUCBUIXQdz78YyC
S1dAp8b7+bXrIv91qjZBietqCc2DjbAt4b2WxJkyZLjVuJlwp0U0cPxuLcATuC+4
dKxFlsDJarvdAGP/b6hFnDImYs+uG3hb02AB3Jbsur63tQM+fFx3bzZocEB8AdV2
gJBZgKTWAAAAAE1FTkSuQmCC
```

The icon is embedded as an MIME-encoded 26x26 pixel PNG file directly into the control directive.

For more information, please see maemo.org documentation.

1.10 Creating your package

In this section we'll cover step by step instructions on how to convert an autotoolised project into a Debianised project and how to generate the binary package based on the Debian control file.

Step by step instructions (with screen captures shortly):

- Clean up your source package (by running **antigen.sh** or similar tool). The important bit here is to get rid of the **--prefix** information after previous testing. Building software packages is done without the **--prefix** option to configure.
- Bootstrap your source by running **autogen.sh**. Fill in the missing files and be especially careful about the **COPYING** file since that holds the software license that will be included in the package. It is GPL by default. For the example code, a special License is used (which is used for all example code of this material) instead of GPL.
- Run the configure script once without any parameters in order to get the make targets.
- Verify software package coherency by making **distcheck** target. Do not continue past this point if **distcheck** fails for some reason.
- Make a **dist** package of your source.
- Create a separate directory under which you'll do your packaging (**packaging** is not a bad name for it) and copy the **dist** package there.
- Change into the directory and extract your distribution package there. This should result in the proper **package-version** subdirectory if your

autotoolisation went correctly. It is important that the path is of the correct format, otherwise the packaging process will not go smoothly!

- Change your directory into the freshly uncompressed source directory(**package-version**).
- Bootstrap the necessary Debian control files:
 - `DEBFULLNAME="Your Fullname" dh_make -e your@email.com -f ../path-to-dist.tar.gz`
Here `../path-to-dist.tar.gz` refers to the `dist` target built source tarball. Making a copy of it under **packaging** is not a bad idea. `dh_make` will ask you two questions interactively:
 - * Answer `s` to the first (single binary package for us).
 - * Press `ENTER` to approve the configuration (if it's not correct, press `Ctrl+c`).
- List contents of `./debian`. It should contain a lot of files.
- Replace the `debian/copyright` file with the contents of your real license file (take a look at the original version first though).
- Remove the ones that you don't need (you need these: **changelog**, **control**, **copyright**, **compat** and **rules**).
- Edit the control file:
 - Check your `Maintainer` name and email (if they're not correct). If you have to change them, you'll also have to update the **changelog**-file since it was generated automatically as well with the same information.
 - Change the `Section` so that the main section will be `user` and the subsection any of the ones listed previously. The example below will use `user/other`.
 - Change `Build-Depends` to contain all the necessary development packages that need to be installed in order to build your package. In our case: **debhelper (>= 5)**, **libgtk2.0-dev**, **libhildon1-dev**, **libhildonfm2-dev**, **libosso-gnomevfs2-dev**, **libgconf2-dev**, **libosso-dev**. To get such a list, start with the library names that you need for `pkg-config`, then find out which Debian packages the libraries come in(`dpkg -l`). You can also try to find the packages owning the `pkg-config` configuration files (all under `/usr/lib/pkgconfig/`) with `dpkg -S /usr/lib/pkgconfig/hildon-1.pc` for example.
 - Modify the `Depends` to read **\$shlibs:Depends** (so that dependencies will be filled by `dpkg-buildpackage` automatically). The **\$misc:Depends** is not normally necessary unless dealing with complex packages. Please see the manual page for `debhelper` if you think you need it.
 - Leave the `Architecture` as is for now (it will be filled by `dpkg-buildpackage` automatically).

- Consider whether you want to add an icon to your package. If you do, please check the relevant documentation on maemo.org on how to generate the icon MIME encoded field.
- Change back to the source directory just above **debian**.
- `dpkg-buildpackage -rfakeroot .`
- The binary package is left at the directory above the current one.
dpkg-buildpackage will also automatically create the source package files for your application (in the same directory as the binary package).

Let's see what will happen when we follow these instructions. We have already tested the autotoolised package before and verified that it works with `make distcheck`. We start by from a clean setup, by doing the `dist` target we'll get the source tarball that we'll need to start with the package building process.

```
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ls -la
total 80
drwxr-xr-x  2 user user 4096 Nov 18 20:37 .
drwxr-xr-x 10 user root 4096 Nov 18 20:36 ..
-rw-r--r--  1 user user  124 Nov 18 20:36 AUTHORS
-rw-r--r--  1 user user 1174 Nov 18 20:36 COPYING
-rw-r--r--  1 user user  159 Nov 18 20:36 ChangeLog
-rw-r--r--  1 user user 1174 Nov 18 20:36 License
-rw-r--r--  1 user user  847 Nov 18 20:36 Makefile.am
-rw-r--r--  1 user user   85 Nov 18 20:36 NEWS
-rw-r--r--  1 user user   73 Nov 18 20:36 README
-rwxr-xr-x  1 user user  706 Nov 18 20:36 antigen.sh
-rwxr-xr-x  1 user user 1024 Nov 18 20:36 autogen.sh
-rw-rw-r--  1 user user 1542 Nov 18 20:36 configure.ac
-rw-rw-r--  1 user user 23910 Nov 18 20:36 hhwx.c
-rw-r--r--  1 user user  162 Nov 18 20:36 hhwx.desktop.in
-rw-r--r--  1 user user   59 Nov 18 20:36 org.maemo.hhwx.service.in
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ./autogen.sh
.. output cut ..
Makefile.am: installing './depcomp'
Ready to go (run configure)
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ./configure
checking for a BSD-compatible install... /scratchbox/tools/bin/install -c
checking whether build environment is sane... yes
.. output cut ..
checking pkg-config is at least version 0.9.0... yes
checking for HHW... yes
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands
configure: creating ./config.status
config.status: creating Makefile
config.status: creating hhwx.desktop
config.status: creating org.maemo.hhwx.service
config.status: executing depfiles commands
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > make dist
{ test ! -d hhwx-0.1 || .. output cut..
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > ls -l *.tar.gz
-rw-rw-r--  1 user user 80229 Nov 18 20:51 hhwx-0.1.tar.gz
```

Preparing the source code for Debian packaging

You might notice that the **antigen.sh** step was omitted above. Since the source code was in a pristine condition, running **antigen.sh** was unnecessary (as running the `make distcheck` -step).

Next, let's create a directory one level up where we'll extract the package and then prepare the extracted source for Debianisation:

```
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > mkdir ../packaging
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > cp hhwx-0.1.tar.gz ../packaging
[sbox-DIABLO_X86: ~/autotoolized-hildon-helloworld] > cd ../packaging
[sbox-DIABLO_X86: ~/packaging] > tar xzvf hhwx-0.1.tar.gz
hhwx-0.1/
hhwx-0.1/COPYING
hhwx-0.1/configure.ac
hhwx-0.1/NEWS
hhwx-0.1/INSTALL
hhwx-0.1/Makefile.in
hhwx-0.1/org.maemo.hhwX.service.in
hhwx-0.1/hhwX.desktop.in
hhwx-0.1/hhwX.c
hhwx-0.1/aclocal.m4
hhwx-0.1/Makefile.am
hhwx-0.1/AUTHORS
hhwx-0.1/README
hhwx-0.1/configure
hhwx-0.1/depcomp
hhwx-0.1/missing
hhwx-0.1/install-sh
hhwx-0.1/ChangeLog
[sbox-DIABLO_X86: ~/packaging] > cd hhwx-0.1
```

Creating the packaging working space and extracting the source

Next, we use the `dh_make` program to create the necessary Debian files so that we don't have to write everything ourselves:

```
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > DEBFULLNAME="User Universal" \
dh_make -e user@maemo.org -f ../hhwx-0.1.tar.gz

Type of package: single binary, multiple binary, library, kernel module or cdb?
[s/m/l/k/b] s

Maintainer name : User Universal
Email-Address   : user@maemo.org
Date            : Sun, 18 Nov 2007 21:06:26 +0200
Package Name    : hhwx
Version         : 0.1
License         : blank
Type of Package : Single
Hit <enter> to confirm: [ENTER]
Done. Please edit the files in the debian/ subdirectory now. hhwx
uses a configure script, so you probably don't have to edit the Makefiles.
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > cd debian
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > ls -l
total 104
-rw-rw-r-- 1 user user 171 Nov 18 21:06 README.Debian
-rw-rw-r-- 1 user user 181 Nov 18 21:06 changelog
-rw-rw-r-- 1 user user  2 Nov 18 21:06 compat
-rw-rw-r-- 1 user user 337 Nov 18 21:06 control
-rw-rw-r-- 1 user user 621 Nov 18 21:06 copyright
-rw-rw-r-- 1 user user  77 Nov 18 21:06 cron.d.ex
-rw-rw-r-- 1 user user  17 Nov 18 21:06 dirs
-rw-rw-r-- 1 user user  12 Nov 18 21:06 docs
-rw-rw-r-- 1 user user 1224 Nov 18 21:06 emacs-en-install.ex
-rw-rw-r-- 1 user user  456 Nov 18 21:06 emacs-en-remove.ex
-rw-rw-r-- 1 user user 1111 Nov 18 21:06 emacs-en-startup.ex
-rw-rw-r-- 1 user user  226 Nov 18 21:06 hhwx-default.ex
-rw-rw-r-- 1 user user  486 Nov 18 21:06 hhwx.doc-base.EX
-rw-rw-r-- 1 user user 2106 Nov 18 21:06 init.d.ex
-rw-rw-r-- 1 user user 1731 Nov 18 21:06 manpage.1.ex
-rw-rw-r-- 1 user user 4640 Nov 18 21:06 manpage.sgml.ex
-rw-rw-r-- 1 user user 4597 Nov 18 21:06 manpage.xml.ex
-rw-rw-r-- 1 user user  109 Nov 18 21:06 menu.ex
-rw-rw-r-- 1 user user  956 Nov 18 21:06 postinst.ex
-rw-rw-r-- 1 user user  929 Nov 18 21:06 postrm.ex
-rw-rw-r-- 1 user user  689 Nov 18 21:06 preinst.ex
-rw-rw-r-- 1 user user  876 Nov 18 21:06 prerm.ex
-rwxr-xr-x 1 user user 2515 Nov 18 21:06 rules
-rw-rw-r-- 1 user user  659 Nov 18 21:06 watch.ex
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > cat ../COPYING > copyright
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > ls -l ../COPYING copyright
-rw-r--r-- 1 user user 1174 Nov 18 20:36 ../COPYING
-rw-rw-r-- 1 user user 1174 Nov 18 21:07 copyright
```

Creating template files for packaging and overwriting the default copyright file.

The **copyright** file was overwritten because the default one (a GPL template) didn't suite the project. You should check the template whether it is suitable for use(`dh_make` also supports couple of other open source copyright/license templates).

Now we need to remove the files that we don't need:

```
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > rm *.ex *.EX dirs docs README.Debian
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1/debian] > ls -l
total 20
-rw-rw-r-- 1 user user 181 Nov 18 21:06 changelog
-rw-rw-r-- 1 user user  2 Nov 18 21:06 compat
-rw-rw-r-- 1 user user 337 Nov 18 21:06 control
-rw-rw-r-- 1 user user 1174 Nov 18 21:07 copyright
-rwxr-xr-x 1 user user 2515 Nov 18 21:06 rules
```

Removing unnecessary example files

Then edit the `control` file to contain the necessary dependencies, the necessary Section and other settings. We're not adding the icon to our package at this time.

```
Source: hhwx
Section: user/other
Priority: extra
Maintainer: User Universal <user@maemo.org>
Build-Depends: debhelper (>= 5), libgtk2.0-dev, libhildon1-dev,
  libhildonfm2-dev, libosso-gnomevfs2-dev, libgconf2-dev, libosso-dev
Standards-Version: 3.7.2

Package: hhwx
Architecture: any
Depends: ${shlibs:Depends}
Description: The ultimate Hello World
  This is the tenth version of Hello World.
  .
  Demonstrates simple GUI things with Hildon and support
  libraries.
```

The modified version of control file

And now everything should be ready for us to build the package:


```

[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > dpkg-buildpackage -rfakeroot
dpkg-buildpackage: source package is hhwx
dpkg-buildpackage: source version is 0.1-1
dpkg-buildpackage: source changed by User Universal <user@maemo.org>
dpkg-buildpackage: host architecture i386
dpkg-buildpackage: source version without epoch 0.1-1
: Using Scratchbox tools to satisfy builddeps
fakeroot debian/rules clean
dh_testdir
dh_testroot
.. output cut ..
dh_strip
dh_compress
dh_fixperms
dh_installdeb
dh_shlibdeps
dh_gencontrol
dh_md5sums
dh_builddeb
dpkg-deb: building package 'hhwx' in '../hhwx_0.1-1_i386.deb'.
dpkg-genchanges
dpkg-genchanges: including full source code in upload
dpkg-buildpackage: full upload (original source is included)
[sbox-DIABLO_X86: ~/packaging/hhwx-0.1] > cd ..
[sbox-DIABLO_X86: ~/packaging] > ls -l
total 216
drwxrwxr-x 4 user user 4096 Nov 18 21:44 hhwx-0.1
-rw-rw-r-- 1 user user 80229 Nov 18 21:00 hhwx-0.1.tar.gz
-rw-rw-r-- 1 user user 24248 Nov 18 21:44 hhwx_0.1-1.diff.gz
-rw-rw-r-- 1 user user 400 Nov 18 21:44 hhwx_0.1-1.dsc
-rw-rw-r-- 1 user user 727 Nov 18 21:44 hhwx_0.1-1_i386.changes
-rw-rw-r-- 1 user user 11750 Nov 18 21:44 hhwx_0.1-1_i386.deb
-rw-rw-r-- 1 user user 80229 Nov 18 21:00 hhwx_0.1.orig.tar.gz
[sbox-DIABLO_X86: ~/packaging] > dpkg --info hhwx_0.1-1_i386.deb
new debian package, version 2.0.
size 11750 bytes: control archive= 773 bytes.
    641 bytes, 13 lines   control
    409 bytes,  6 lines   md5sums
Package: hhwx
Version: 0.1-1
Section: user/other
Priority: extra
Architecture: i386
Depends: libatk1.0-0 (>= 1.12.2), libc6 (>= 2.5.0-1), libcairo2 (>= 1.4.10),
        libdbus-1-3 (>= 0.94), libdbus-glib-1-2 (>= 0.74), libgconf2-6 (>= 2.13.5),
        libglib2.0-0 (>= 2.12.12-1osso3), libgtk2.0-0 (>= 2:2.10.12-0osso15),
        libhildon1 (>= 1.0.11), libhildonfm2 (>= 1:1.9.46), libosso-gnomevfs2-0,
        libosso1 (>= 2.13), libpango1.0-0 (>= 1.16.4)
Installed-Size: 84
Maintainer: User Universal <user@maemo.org>
Description: The ultimate Hello World
 This is the tenth version of Hello World.

.
Demonstrates simple GUI things with Hildon and support
libraries.
[sbox-DIABLO_X86: ~/packaging] > dpkg --contents hhwx_0.1-1_i386.deb
drwxr-xr-x root/root      0 2007-11-18 21:44:44 ./
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/
drwxr-xr-x root/root      0 2007-11-18 21:44:44 ./usr/bin/
-rwxr-xr-x root/root 20384 2007-11-18 21:44:44 ./usr/bin/hhwx
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/applications/
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/applications/hildon/
-rw-r--r-- root/root    158 2007-11-18 21:44:43 ./usr/share/applications/hildon/hhwx.desktop
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/dbus-1/
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/dbus-1/services/
-rw-r--r-- root/root     55 2007-11-18 21:44:43 ./usr/share/dbus-1/services/org.maemo.hhwx.service
drwxr-xr-x root/root      0 2007-11-18 21:44:43 ./usr/share/doc/
drwxr-xr-x root/root      0 2007-11-18 21:44:44 ./usr/share/doc/hhwx/
-rw-r--r-- root/root    1174 2007-11-18 21:07:54 ./usr/share/doc/hhwx/copyright
-rw-r--r-- root/root    158 2007-11-18 20:36:56 ./usr/share/doc/hhwx/changelog.gz
-rw-r--r-- root/root    189 2007-11-18 21:06:40 ./usr/share/doc/hhwx/changelog.Debian.gz

```

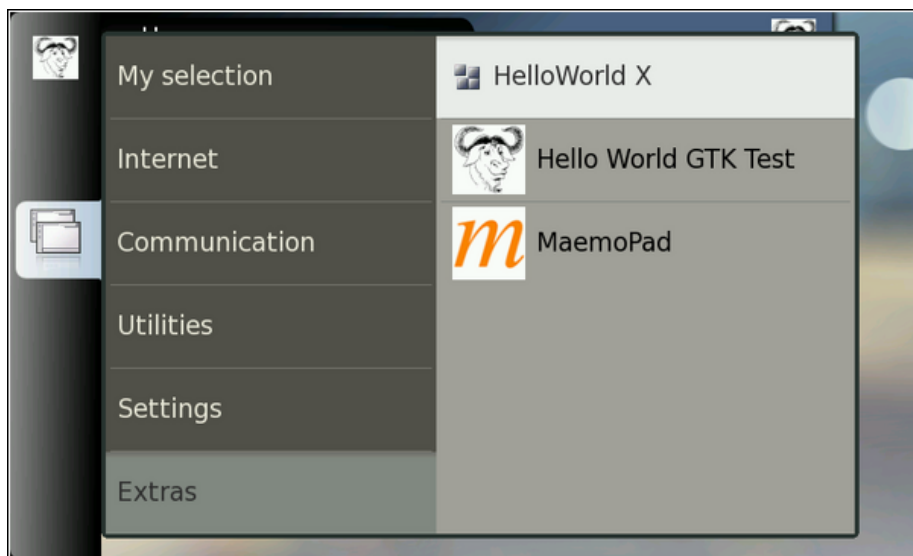
Building the package and listing its information and contents

Now we're ready to install the package using `dpkg`.

```
[sbox-DIABLO_X86: ~/packaging] > fakeroot dpkg -i hhwx_0.1-1_i386.deb
Selecting previously deselected package hhwx.
(Reading database ... 19483 files and directories currently installed.)
Unpacking hhwx (from hhwx_0.1-1_i386.deb) ...
Setting up hhwx (0.1-1) ...
```

Installing the package manually

After the installation, the menu should now be updated and you should be able to start the application.



Everything seems to be in order

We then also remove the package manually.

```
[sbox-DIABLO_X86: ~/packaging] > fakeroot dpkg --purge hhwx
(Reading database ... 19490 files and directories currently installed.)
Removing hhwx ...
```

Removing the package manually.

You can verify that the application has been removed from the menus now.

End users normally will not use `dpkg` manually, but instead they will use the Application manager program in the target device. In order for the Application manager to find the file to install, it will need to be located under `/MyDocs/`. You then need to use the "install from a file..." function from the menu.

1.11 Adding debugging support for a package

So that your application can be debugged by others, you should consider adding a separate package for your application which will contain the debugging information. Normally debugging symbols are stripped as part of

the package building process above, so the resulting package ends up being without debugging support. This is an issue on the ARMEL target.

In order to create a debugging version of your package, please see the "Making a Debian Debug Package" section of Maemo Diablo Reference Manual.

1.12 Building the package for a device

At this point you should make sure that you have the ARMEL target in your Scratchbox as well as the X86 one which you've (hopefully) been using so far:

```
[sbox-DIABLO_X86: ~] > sb-conf ls -T
DIABLO_ARMEL
DIABLO_X86
```

Listing set up targets in Scratchbox.

If you only have the X86 target setup, you will need to setup the ARMEL target now, before proceeding. Please see the last chapter of "maemo Getting Started" material for instructions. If you used the automatic installation script, you should automatically have both targets.

If you have your software, or the AF running, you'll need to stop both now(`af-sb-init.sh stop` to stop the AF). It is important to do this before switching targets, although sbbox normally won't allow you to switch anyway if you have any processes running in the current target.

When you're done with shutting down the current target, switch your target to ARMEL:

```
[sbox-DIABLO_X86: ~] > sb-conf select DIABLO_ARMEL
.. screen clears ..
[sbox-DIABLO_ARMEL: ~] > arch
arm
```

Switching the Scratchbox target.

To build the Debian package in the ARMEL target:

- Rebuild your Debian package(`dpkg-buildpackage -rfakeroot`), which should result in an `armel`-version deb-file.
- Install the `armel`-version of Debian package (it's a separate target after all). You'll need to use `dpkg -i` for this. Also check that your package is removable(`apt-get remove`).

One noteworthy thing about the ARMEL target is that it's not really meant to act as a testing environment. How much of your software will work, will depend on the version of Qemu that will be used to emulate the ARM instructions. In short, it's best to test the ARMEL versions of your packages on a real device.

1.13 Installing packages into an Internet Tablet

Once you've built the ARM-version of your package, you'll next need to transfer it to the device. Since the device appears as an USB mass storage device to your

Linux, it is normally enough to attach the USB cable and then mount the device into some suitable directory under Linux. Also most modern graphical environments will do the mounting step for you, so don't be surprised if you don't need to do the mounting manually.

The storage that you can access this way is restricted to the memory card inside the device. It is not possible to access the internal flash memory of the device in this way. The filesystem on the memory card will be VFAT, so it will contain all the normal problems that are related to Windows file-names. Copy the package to the mounted directory and then un-mount the USB storage(`pumount` is normally enough, or you can use the graphical interface). After detaching the cable from the device the device will mount the card internally and it can access the contents on the card.

Because the storage space is not available to both the device and your Linux desktop at the same time, some people opt to use an SSH server and use `scp` to copy files directly. Setting up an SSH server on the device is not covered here, but is pretty simple. Just remember to set your device into "R&D"-mode.

Once the package file is on the device (under `/home/user/MyDocs`), use the Application manager and select the package file to install. If something will go wrong, you'll probably need a command line access to the device. Use the supplied X-Term emulator, and `sudo gainroot` to get root privileges (device needs to be in "R&D"-mode for `gainroot` to work).