Pdaxrom: Create An Ipk Howto

From OESF

Table of contents

directories

To create an ipk the first thing we have to do is to create a directory tree that contains the files that will be installed (and only these files).

For instance if we have one exectuable foo that should go in /usr/bin and a file foo.conf in /etc. We want to have the following structure:

```
ipkg-temp/
|-- etc
|   `-- foo.conf
`-- usr
    `-- bin
        `-- foo
```

NB: You should use a Linux partition to hold ipkg-temp. If you use an msdos/fat/vfat partition then the permissions in the resulting package will be completely messed up.

Installing the application after a compilation

Take care when you do a make install. If you are compiling on your zaurus you might easily fill the / root partition. If you are cross compiling you might install the application on your desktop pc (and possibly break something).

It's a good idea to redirect the output of the installation to a log file just to be sure you can find what

files has been placed where:

```
make install 2>&1 > installation.log.
```

DESTDIR

Makefiles created by the classic "./configure" script often take into account the variable DESTDIR. If set "make install" will install the application under the directory pointed by DESTDIR rather than in /. To set it you can for instance type (prefer the full path to a relative one):

```
make DESTDIR=/home/ubuntu/links-2.1pre20/ipkg-temp/ install
```

But not all makefiles take DESTDIR into account, to check this I usually do:

```
grep DESTDIR Makefile
```

If it returns something, then there is a good chance that DESTDIR is valid.

Another possible trick

A trick I use on the zaurus is to mount a partition over /usr/local (in the case where I know that my application will go into /usr/local using the --prefix= argument for example):

```
mkdir -p /mnt/card/ipkg-tmp/usr/local
mount -bind /mnt/card/ipkg-tmp/usr/local /usr/local
make install 2>&1 > installation.log
umount /usr/local
```

Now my application is now installed in /mnt/card/ipkg-tmp and the content of /usr/local is not modified.

Stripping the binaries

Before releasing an ipk it's better to strip the binaries. That is the executables under /bin but also the shared libraries .so. Just run:

```
arm-linux-strip myexecutable
```

this will reduce the size of your binaries.

Stripping is an important step as the space saved by this operation can be significant.

Sometimes the Makefile provides an install-strip target, that will strip the binaries for you. You might want to try to use make install-strip instead of the classic install.

Removing the documentation

Generally your application comes with its documentation. You might want to remove the man pages, info pages from your ipk. It's up to you to choose to remove the documentation or not.

Try to remove what you think is not important. Not all the zaurus owners have a 4GB micro drive.

You can also create 2 separate packages, one with the application, one with the documentation. Leaving the decision to install or not the documentation to your users.

Libraries

When you compile a library, in general you can put in your ipk the stripped .so.x.x files and remove the rest. (.h .a .la ....). This is in most cases enough for the applications that uses the library.

However it is a good thing if you take the time to make a second ipk that contains all the files (header files .h, static libraries ..).

Name this second ipk by adding a "-devel" suffix to the package name. This ipk will be useful to someone who wants to compile an application that depends on your library.

Create a shortcut in the menu

If you want to add a shortcut in the menu and on the desktop you should add a file foo.desktop (foo being the name of our application). The .desktop file look like this:

```
[Desktop Entry]
Exec=emacs -fn 8x13
Icon=emacs.png
Terminal=false
Type=Application
Categories=Application;Office;WordProcessor;X-Red-Hat-Base;
StartupNotify=false
Encoding=UTF-8
Name=Gnu Emacs
```

The 3 most important fields are:

- Exec which specify the command that will be executed,
- Name which specify the label that will appear next to the icon,
- Icon which specify the icon file for the icon. If no path is specify you should put the icon in /usr/share/pixmaps

For our foo package our directory tree will look like this:

```
ipkg-temp/
|-- etc
|   `-- foo.conf
`-- usr
    |-- bin
    |   `-- foo
    `-- share
        |-- applications
```

```
|   `-- foo.desktop
`-- pixmaps
    `-- foo.png
```

If you want to add a shortcut for rox, look in /usr/apps/.

The control file

Next we need to write a file named control. This file will describe our applications. Each line of this file contains entries of the form:

```
field : value
```

You should place this file in a directory CONTROL together with the other files. Like this:

```
ipkg-temp/
|-- CONTROL
|   `-- control
|-- etc
|   `-- foo.conf
`-- usr
    |-- bin
    |   `-- foo
    `-- share
        |-- applications
        |   `-- foo.desktop
        `-- pixmaps
            `-- foo.png
```

A pretty minimal control file will look like this:

```
Package: lighttpd
Version: 1.4.8
Architecture: armv5tel
Maintainer: pierre.gaston@gmail.com
Description: fast and light http server
```

Package

This field is the name of your package, typically the name of your application. It should only contain alphanumeric characters or '-'.

Don't put an underscore in the package name, the underscore is used to separate the different parts of the filename of the ipk. For instance if you want to make a doc package use a name like foo-doc and not foo_doc

Version

Version of your package. Generally reflects the version of the application. Don't put a _ in it.

Architecture

Use armvtel for xscale, arm otherwise. Maybe something like armvetil if you compiled for pxa270, though no convention seems to exist at this time...

Maintainer

Put your e-mail address rather than the one of the creator of the application. Your address is perhaps more interesting because if someone wants to compile a newer version of the application you might be more helpful than the original developer.

Description

Some words that describe the application. Remember that the description should stay on the line starting with "Description:". It can be a long line but it must be one line. Don't press press Enter while writing it.

Depends

This field should be a list separated by whitespace of package names. ipkg will check that the packages listed here are installed before installing your application, and automatically install the missing ones.

One way to check the dependencies is to use "ldd" to see the list of shared libraries that are used by your application. You can then use "ipkg search libfile.so" to find the packge that cotains libfile.so.

For instance if I want to find the dependencies of mc I can do:

```
# ldd /usr/bin/mc
        libglib-2.0.so.0 => /usr/lib/libglib-2.0.so.0 (0x40020000)
        libiconv.so.2 => /usr/lib/libiconv.so.2 (0x40091000)
        libSM.so.6 => /usr/X11R6/lib/libSM.so.6 (0x40172000)
        libICE.so.6 => /usr/X11R6/lib/libICE.so.6 (0x40182000)
        libX11.so.6 => /usr/X11R6/lib/libX11.so.6 (0x4019f000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x40265000)
        libc.so.6 => /lib/libc.so.6 (0x40278000)
        libdl.so.2 => /lib/libdl.so.2 (0x40398000)
```

```
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
# ipkg search libICE.so.6
xfree: //usr/X11R6/lib/libICE.so.6.3
xfree: //usr/X11R6/lib/libICE.so.6
# ipkg search libglib-2.0.so.0
glib2: //usr/lib/libglib-2.0.so.0
glib2: //usr/lib/libglib-2.0.so.0.800.1
# ....
```

and I will add :

```
Depends: xfree glib2
```

And so on.

Please note that you no longer have to do this by hand! Please see the Automatically Determining Dependancies (*http://www.oesf.org/forums/index.php?showtopic=18410*) forum post. [Please be sure that the line in your control file starts with `Depends:`, and not some variant of that.]

Other fields

You can add other fields to your control file, for instance a "Source: http://foo.org"

preinst, postinst, prerm, postrm

You can add are four scripts to your package:

- preinst: this script will be executed before the installation, ie before the files are copied over.
- postinst: this one will be executed after the installation but before ipkg-link ie the files will be present in the destination directory but not the symbolic links in /
- prerm: this one will be executed before the files are removed but after ipkg-link ie the files will be present in the destination directory but the symbolic links in / won't exists anymore
- postrm: this one is executed after the files are removed

Place these scripts in the CONTROL directory, together with the control file.

You can use the PKG_ROOT variable in these scripts to get the destination directory where the package is/was/will be installed.

Finally: the ipk

Creating the package

Don't try to make the ipk by hand. Use the mkipkg (*http://mail.pdaxrom.org/contrib/misc/mkipkg*) provided here (*http://mail.pdaxrom.org/contrib/misc/mkipkg*). The script is also provided in both the

cross and native sdk but it is not compatible with some recent versions of tar.

mkipkg will uncover the possible errors in your control file or elsewhere. If mkipkg fails double-check your control file.

When you have a tree structure like:

```
ipkg-temp/
|-- CONTROL
|   `-- control
|-- etc
|   `-- foo.conf
`-- usr
    |-- bin
    |   `-- foo
    `-- share
        |-- applications
        |   `-- foo.desktop
        `-- pixmaps
            `-- foo.png
```

just do:

```
mkipkg ipkg-temp
```

and the ipk will be created for you.

The package

An ipk package is tar.gz a file named:

```
packagename_version_architecture.ipk
```

it contains three files:

- control.tar.gz which contains the control file and the pre post scripts
- data.tar.gz which contains the files of the application
- debian-binary is a text file with 2.0 inside
