HOWTO build an ipkg (including HOWTO on using I-Linux 4.0)

HOWTO - ipkg building

This document describes how to build packages for use with ipkg. ipkg is a utility for the familiar Linux distribution which performs functions similar to apt-get under Debian. However, ipkg is a lighter-weight application, meant for embedded Linux applications. It was originally designed for the iPaq, but it is being used more often these days in various embedded Linux configurations as a general package management tool.

1) Setup

To build ipk's (ipkg binary packages), you will need to obtain versions of the ipkg utilities. These utilities can be obtained by pulling down the CVS repository from the familiar project:

Login to the CVS:

```
    cvs -d :pserver:anoncvs@cvs.handhelds.org:/cvs login
    Use "anoncvs" for the password
```

Obtain the ipkg tools

```
    cvs -d :pserver:anoncvs@cvs.handhelds.org:/cvs co familiar/dist/ipkg/util
```

This will create the directory trees familiar/dist/ipkg/util.

This retrieves several shell scripts which are important in the building of the ipkgs. The function of the most important tools are listed below:

ipkg-build	Builds ipk's from a given directory structure and control files.
ipkg-buildpackage	Takes source code, a set of control files, and a build rule file to compile the source, install it, and create the ipk from it. Requires ipkg-build
ipkg-make-index	Takes a bunch of ipk's and outputs to stdout the contents of the Packages file that would be created.

Next, if you are on a multiuser system, you will want to make a few changes so that ipkg-build* uses ~/tmp rather than the system /tmp. (This can solve some aggravating problems if someone is creating a package and stopped the build - namely, the inability to create a package using the same directory). This is necessary if you wish to build the packages contained in the I-Linux source. Apply the ipkg-buildpackage.patch patch to your ipkg-buildpackage in this case:

```
    cd <Location of script>
    patch -p2 < ipkg-buildpackage.patch
```

Modify the SCRIPTDIR line in ipkg-buildpackage to reflect the location where you installed your script as well.

The rest of the shell scripts are utilities for working with ipk files.

In addition, it is highly recommended to install the fakeroot(1) program, which is a program that allows regular uses to pretend that they are root in order to be able to do things that often require root, but don't require root permissions. Such tasks include
  -- Making device nodes
  -- Changing ownership of files

  -- Changing group ownership

Note that fakeroot(1) does not give a user actual root permissions, it merely
enables a user to do these tasks in a special shell. Regular files are created
on the filesystem when device nodes are created, and ownership data is stored
only for the shell session. A fakeroot(1)ed user has no more permissions than
he or she has during a regular session. Note that also the device nodes etc
yA fakeroot(1)ed user has no more permissions than
he or she has during a regular session. Note that also the device nodes etc
you create only last for a fakeroot(1) session - it isn't persistent.
However, tar and similiar utilties are 'fooled' into thinking that the
files are special files and retain the information like device nodes, owners
and group, etc.

2) Building ipk's from source

To build an ipk package file from source, we'll use the hello-world
application. This application just does the standard "Hello, world!" greeting.
Not very useful, but it illustrates the various aspects of building the
package (as well as being simple enough that the actual program itself can
be ignored).

The completed and working hello world project is available here to which
those who want to see the final results can simply inspect the code.

But assuming we want to begin from scratch, perhaps just after unpacking the
source tarball, one would proceed as follows:

  -- Learn the build process
  -- Write the control file
  -- Write the rules file
  -- Write optional scripts
  -- Create the package

We'll go through each of these steps in turn.

2.1) Learn the build process

Every application is different. Most can get by with a simple

```
   ./configure
   make
   make install
```

but this is often not the case if you are cross-compiling the packages (i.e.,
you often have to specify prefix directories, architectures, etc). Thus, one
should understand the nuances of building the specific package (i.e., what
options to pass configure, what variables to set to do a make, and what's the
prefix used so make install will do the install inside a directory which will
represent the final filesystem layout) first before proceeding. All the
utilities to build the packages are shell scripts, and can fail with bizarre
error messages (if they do fail), or silently continue on after encountering
an error condition.

For our hello world program, we'll make a new directory and pretend that
we've just unpacked it.

```
  mkdir ~/hello-world
  cp hello.c ~/hello-world
  cd ~/hello-world
```

Now, the build is simply calling the compiler with the right options.

```
  arm-linux-gcc -Wall -g -o hello hello.c
```

To install it, we copy hello to a suitable location on the filesystem, e.g. /usr/bin.

To clean up we delete hello and any .o files that were created (none).

Thus we note this down and we're done.

2.2) Write the control file

The control file is an important file to the packaging tool. It describes every aspect of the program. It tells the packaging tools what dependencies there are in the program (e.g., requires libc6, etc), what the version of the program is (for versioning), what the name of the program is, maintainer, description, etc. The name of this file is, surprisingly enough, "control".

ipkg-buildpackage expects the control file, as well as any other related script files to be located in a subdirectory called ipkg. Thus we need to create it.

```
  cd ~/hello-world
  mkdir ipkg
```

Now, we go into ipkg, and edit the control file.

```
  cd ipkg
  editor control
```

The syntax of the control file is simple. It's basically <tag>: <value>.
If <value> spans multiple lines, each additional line must be indented by one space. The colon is very important in separating out the tag from the value.

The essential tags are, with the values we're using in parentheses:

```
Package         - Package name (hello-world)
Priority        - Is this package important in the image? Optional?
                  (most of the packages will be optional) (optional)
Version         - What version is this package? (0.0.2)
Section         - Where do we put this package in the list? (base)
Architecture    - What CPU architecture does this compiled package run on? (arm)
Maintainer      - Who created the package (i.e., who took the source code
                  and makde this package?). (your name here)
Depends         - What packages does this one depend on (none)
Description     - What is the description of this package? Note that there is
              both a short and long description - the short description
              (approximately 60 characters) is essentially the first line of
              the description (i.e., the value immediately following the
              Description: tag). The long description follows immediately
              after, on the next line, indented by one space.
```

Here is the control file that we're using for hello-world:

```
-----cut-here-File=control-----
Package: hello-world
Priority: optional
Version: 0.0.2
Section: base
Architecture: arm
Maintainer: James Ng <jng@intrinsyc.com>
Depends:
Description: A simple "hello world" application.
 This is used to show how to build ipkg packages from source
```

```
-----end-cut-----
```

2.3) Write the rules file

The rules file describes to ipkg-buildpackage (and ipkg-buildpackage only)
how to take the source code, build it, install it in a temporary location,
and clean up. ipkg-buildpackage calls the file as:

```
  rules [build | install | clean]
```

Thus, you can write a C program, use shell scripts, or my personal favorite,
use make and abuse the POSIX shebang syntax :). The filename is "rules".

The rules file must be executable - the ipkg-buildpackage calls it with the
parameter describing which stage to perform. The rules file exists in the
ipkg/ subdirectory.

The stages are:

```
build   - Take the source code, and compile it
install - Take the compiled source code, and install it to a temporary root
          filesystem.
clean   - Clean up
```

Now, ipkg-buildpackage calls "rules build" and "rules install", then copies
over the various control files and scripts necessary, calls ipkg-build to
create the ipk file, then calls "rules clean" to clean up. Your clean stage
does *NOT* have to remove the directory that you installed your application
in.

For hello, recall from 2.1 what each stage needs to do.

```
build)
        arm-linux-gcc -Wall -g -o hello hello.c

install)
        make filesystem root directory
        make usr/bin under that
        copy hello to usr/bin

clean)
        remove hello (remember we built it in this directory)
        remove the build stamp
```

Thus, our rules file looks as follows

```
-----cut-here-File=rules-----
#!/usr/bin/make -f
#
# $Id: rules.example,v 1.1 2001/07/26 15:36:36 oku Exp $
#
# rules for hello

# Make sure PACKAGE reflects your package name (in control). If it's
# different, it'll break.
PACKAGE=hello-world
# Do not touch this. This points to the "filesystem root" where you'll
# be putting your files (e.g. $(DESTDIR)/usr to put stuff in /usr, etc.
DESTDIR=$(HOME)/tmp/$(PACKAGE)
STRIP=arm-linux-strip --remove-section=.comment --remove-section=.note --strip-unneeded

build: build-stamp
```

```
build-stamp:
# Rules to compile your program go here. You can call the compiler, or
# you can do a configure, make.
        arm-linux-gcc -Wall -g -o hello hello.c
        touch build-stamp

install: build
# Rules to install your program go here. You can manually install it (like this)
# or go and call make install. Please note that you MUST use $(DESTDIR) -
# and create your filesystem directory structure from there. Otherwise
# ipkg will NOT pick it up!
        rm -rf $(DESTDIR)
        mkdir -p $(DESTDIR)/usr/bin
        cp hello $(DESTDIR)/usr/bin/
        $(STRIP) $(DESTDIR)/usr/bin/hello

clean:
# Clean up goes here
# This is called after making and installing the packages, and after the
# ipk is made. Use it to clean up after yourself (removing *.o's, calling
# make clean, etc).
        rm -f build-stamp
        rm -f hello
-----end-cut-here-----
```

Several important notes:
-- Remember to chmod +x rules!
-- ipkg-buildpackage will expect your filesystem root to be ~/tmp/<package name>
   where <package name> is what you defined in the control file. If they're
   not the same, you'll get an empty package.

2.3) Write the optional script files

There are 4 optional script files that can be run at strategic times during installation and removal, and one optional file describing configuration files. These files exist also in the ipkg/ subdirectory.

The file "conffiles" lists, using absolute paths, the location of every configuration file available. So if you have a package that has a configuration file placed in /etc named foo.conf, and another file placed in /usr/local called foo2.conf, your "conffiles" file will contain

```
/etc/foo.conf
/etc/foo2.conf
```

The 4 optional scripts run at strategic times during the install and removal of packages. These can be used to initialize the initial environment, do some post installation fixups, etc. The four scripts are preinst, postinst, prerm, and postrm.

The scripts are executed in the following order, starting with the appropriate ipkg command

```
> ipkg install <package>

... ipkg searches package lists for <package> and downloads the appropriate
    file ...
... ipkg unpacks ipk file ...
preinst script (if it exists) is run
... ipkg unpacks the data tarball ...
postinst script is run (if it exists)
```

```
> ipkg remove <package>

... ipkg checks ...
prerm script is run (if it exists)
... ipkg removes all the files in the package from the filesystem ...
postrm script is run (if it exists)
```

These scripts have to be executable and can be anything - programs, shell scripts, etc. The only requirement is that enough is provided to actually run the script (e.g., you shouldn't make these scripts makefiles, because make isn't installed in most cases, unless you can guarantee that it is in your package).

Demonstration files are provided in the hello-world demo package that simply echo what script is called. Simply build the hello-world demo package and try ipkg install hello-world, and ipkg remove hello-world.

The main reason to use these scripts is to change the way the system works - when you install certain types of files, you have to update the way the system works - vim for example alters several symlinks so /usr/bin/editor points to it. Another common procedure is to install libraries, at which point the program ldconfig must be run to finish off the installation and update the dynamic loader cache. Or kernel modules have to have their dependencies re-checked.

You would do these things both in the postinst script (so just after unpacking, you make the libraries available), as well as the postrm script (so after removing, there isn't a trace left of the old files). Sometimes though, the prerm script would be the place to put the commands (such as altering the symlink to point to another program) before removing the program so that the symlink works throughout the install process.

2.5) Build the package

Finally, we come to actually building the packages!

In the directory where you put your source code and ipkg files, run the complicated commands shown below to actually build the package. Remember, it took a lot of work to get here, so it must take a lot of work to build the final package from source!

```
  ipkg-buildpackage
```

If nothing went wrong, you shouldn't have any errors and you should end up with an ipk file one directory up.

3) Using ipkg-build

If you have a binary package, it's easy to convert it directly. The control file, plus any optional scripts (preinst, postinst, prerm, postrm), and optional control files (conffiles) are placed in the CONTROL subdirectory in the filesystem root. There is no need for a rules file since there is nothing to build.

ipkg-build takes one required parameter and has one optional parameter. The required parameter is the directory which has the filesystem, and the optional is where to put the produced ipk. If you are in the root of the filesystem image, then you can do a

```
  ipkg-build . ..
```

which will take the CONTROL subdirectory, package up the binaries and produce the ipk file in the parent directory.

4) Using ipkg-make-index

Once you have all your ipk files gathered up, you have to create the Packages file which lists all the packages, their dependencies and other information. ipkg and other utilities will download the Packages file and use it to determine what can be installed (i.e., it's a catalog of all the packages available from a feed).

First, you have to place your files on your feed. Get every file you want to serve up that isn't simply a mirror of another feed into on directory. Then change to that directory, and do a

```
  ipkg-make-index . > Packages
```

And messages stating that it's generating the index for each package should be displayed. As ipkg-make-index writes the package information out to stdout, this can be used to append other directories to one Package file (and thus, one feed). Note that ipkg-make-index must be run from the directory the base of the feed is in - the other directories must be accessed relative to the current directory.

5) I-Linux 4.0 Notes

Most of the ipk file building is automated through the use of makefiles and shell scripts. In fact, the main root makefile (the one in the binonly/ and src/ directories traverse a list of subdirectories and calls each of them with the following:

```
  make build
  make copy
  make clean
```

where build is used to compile the packages, copy is used to copy the resulting ipk file from whereever it is to a predefined variable OUTPUTDIR,

The BlueZ package requires access to kernel sources - thus it's readme file should be consulted.

To add a package so it can be automatically built, add a directory either the binonly directory (for binary-only packages), or the src direcotry (for source packages). Add a Makefile that has rules for build, copy and clean - most of the time, if it was done right, you'd have a makefile similar to:

```
-----cut-here-File=Makefile-----
default: build copy clean

build:
        ipkg-buildpackage

copy:
        mv ../*.ipk $(OUTPUT_DIR)

clean:

-----end-cut-here-----
```

Then modify the main makefile, and add your directory to the list of packages. That's all there is to integrate it in - when you go back to the familiar/ directory and do a make ipks, your directory will be traversed and the ipk build with the rest and tossed into the output directory, ready for index making.

6) Converting to/from debian

One of the goals of the Familiar project was to be binary compatible with the ARM Debian port. This includes being able to use the Debian deb files in the port and run them on Familiar. However, the ipk and deb fileformats differ slightly - an ipk is a tarball, while a deb is an ar archive. The contents of each are the same - three files, "debian-binary", "data.tar.gz" and "control.tar.gz".

To mechanically convert a Debian deb file to an ipk file, simply execute the following commands:

```
  ar x <package>.deb
  tar cvzf <package>.ipk ./debian-binary ./data.tar.gz ./control.tar.gz
```

(Although, it's preferable to actually edit the data.tar.gz file (it's the programs as they'll appear in the filesystem) to remove things like docs and manpages, and to strip binaries, then just repack it up as data.tar.gz).

Converting an ipk to deb is a less trivial problem. While there's probably a way to go directly between an ipk to deb, attempts have failed. Thus, the quickest way is to unpack the ipk file - create a temporary directory, unpack data.tar.gz into it, unpack control.tar.gz into a subdirectory named DEBIAN, and run dpkg-deb -b <temporary directory> to build the deb file. You should use fakeroot while doing this.

 $Revision$ $Author$ $Date$