

# pWare Packaging Guide

William Jojo - Hudson Valley Community College

(Revision: 20080606 [DRAFT])

## Abstract and Governance

This document shows a simplified approach to packaging applications that are intended to be used with pWare. It must be remembered that at no point should any pWare packages rely on packages from external projects - not that any of those packages or projects are necessarily bad. Rather, the intent is to keep the pWare environment as clean as possible with dependencies provided from within.

If you need one or more products, libraries or modules to fill in a requirement or desired feature of your proposed product, build those dependencies first and package them. Of course, you should check the pWare site first for availability as new support libraries are added all the time. Now build your intended product and package it with the noted dependencies at the appropriate levels.

Do not give up when a product fails to build. AIX can be very tricky to make cooperate with certain programming styles, standards and selection of libraries. In addition, you may have to hunt down system/library calls that would otherwise exist in **libc.a** on another platform.

Whenever possible, always build for shared libraries and decline static ones if the product offers both. It is expected that a reasonable level of testing of all executable components and libraries will be performed before submission to the pWare suite of products.

Only packages in bff format (built with **mkinstallp**) will be accepted.

Packages that clobber existing configuration files will not be accepted.

A package will also be denied if any one executable, shared library or shared member fails to resolve all of its dependencies when the package is selected by itself. These issues can often be resolved by the use of **ldd** and **dump -n**. It is then a simple matter of updating the package dependencies.

It is acceptable to have components of a product that are intended for another product without a stated dependency. For example, **svn** has two modules for **Apache**, but **Apache** is not a dependency of **svn**. Nor is **svn** a dependency of **Apache**. The user is granted the opportunity to use **svn** without installing **Apache** and vice-versa.

It is expected that **LIBPATH** is unset in production environments running pWare products. Setting **LIBPATH** will override the embedded **libpath** within the application thereby (in most cases) breaking it. A package that needs to set **LIBPATH** will be denied. Rework the **-blibpath** options to make the product work.

The tools provided in this tutorial are intended to build packages sequentially. There is never a guarantee that packages will cooperate with the automated configuring, building and packaging of a product. However, more than 70% of the packages in pWare can be setup for unattended, automatic packaging.

Keep copious notes on your endeavor(s). They will never let you down after a month has passed and it is time for an upgrade. If you are uncomfortable with producing a package or just wish to offer a build solution, simply pass them along - they will gladly be accepted and I am more than willing to build the package.

One last note: I use AIX 5300-06-03 as my baseline for building packages. My production servers run a mix of 5300-06-04, 5300-07-02 and 5300-07-04. I have done limited testing on 6100-00-03 and 6100-01-01. If you have questions, always post to the list.

## The Tools

You should use the (minimum) following tools provided at the web site and install media:

```
pware53.autoconf.rte
pware53.base.rte
pware53.bash.rte (helpful in speeding up builds)
pware53.bison.rte
pware53.diffutils.rte
pware53.flex.rte
pware53.gcc-g++.rte (4.2.3)
pware53.gdb.rte
pware53.gettext.rte
pware53.libiconv.rte
pware53.m4.rte
pware53.make.rte (3.81)
pware53.patch.rte
pware53.popt.rte
pware53.readline.rte
pware53.tar.rte
pware53.texinfo.rte
pware53.unzip.rte
pware53.zip.rte
pware53.zlib.rte
bos.adt.insttools
```

The list above should provide all of the basic necessities for building products. Of course, you will likely need more for larger projects.

## The Staging Area

These scripts and tree are how I produce packages. It is clean, simple and fast. This is provided here in the hope that it will help you too, or, if nothing else, will get another set of eyes upon the task to make it even better than it is today. The scripts are tied to the staging area layout like so:

```
/stage/
|-- lpp/
    |-- product1/
        |-- deps
        |-- product1.filelist
        |-- product1.template
        |-- opt/
            |-- pware/
                |-- bin/
                |-- lib/
                |-- etc/
    |-- log/
    |-- product1/
        |-- doit
        |-- product1-1.1.2/
    |-- product2/
        |-- product2-0.5/
```

The **/stage** filesystem need not be more than 20GB which holds everything that is packaged to date. The **lpp** subdirectory is the packaging stage. All other directories refer to the product stage. The **doit** file is a simple script to perform the package configuration for building the package and the **deps** file is a list of package dependencies.

The **log** directory holds the logs for each build and package.

When the package is complete, there will be a complete installation in **/opt/pware** and in **/stage/lpp/package/opt/pware**.

## Building the Package

Both the **doit** and **deps** files are used by the **build-lpp** script. In addition, there is an **lpp-file** that is used as an argument to **build-lpp**. The **build-lpp** script will generate the **product.filelist** and **product.template** files. Each file is demonstrated below and is further documented here:

[http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/pkging\\_sw4\\_install.htm](http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.genprogc/doc/genprogc/pkging_sw4_install.htm)

### deps

The dependency file is a list of dependencies as stated in the previous document. Typically prerequisites (**prereq**) are all that are necessary. This list typically looks something like the following:

```
*prereq pware53.base.rte 5.3.0.0; *prereq pware53.openldap.rte 2.3.39.0; *prereq
pware53.openssl.rte 0.9.8.7; *prereq pware53.cyrus-sasl.rte 2.1.22.0;
```

It is simply one long run-on dependency list with no line breaks. Each begins with an asterisk (\*) and are separated with a semicolon (;).

### doit

The **doit** file is a script that generally runs **configure** for a given package. It should be placed in the root of the product directory above any specific product version(s). Most of the ones used for pWare look something like this:

```
env CC="gcc -O2 -Wl,-blibpath:/opt/pware/lib:/usr/lib:/lib,-brtl" \
CFLAGS="-I/opt/pware/include" \
LDFLAGS="-L/opt/pware/lib" \
./configure --prefix=/opt/pware \
--enable-shared=yes --enable-static=no \
--enable-log-host=yes \
--with-aixauth -with-blibpath=/opt/pware/lib:/usr/lib:/lib \
--with-fqdn --with-ldap=/opt/pware
```

The obvious benefit is natural documentation that you can reuse when a new version of a product

comes out.

## **lpp-file**

This is what I call the file, but it really can be called anything. It is a sequential package file. The format of each line is the following:

```
package-name | version | path-to-package-source | path-to-doit | display-name
```

A line that starts with a pound symbol (#) is treated as a comment. This way you can leave previous build lines present and uncomment when you need to rebuild a product at a current or future level.

Below are some examples from a typical **lpp-file**:

```
#jpgtn|2.6.0.0|/stage/jpgtn/jpgtn-2.06|/stage/jpgtn/doit|jpgtn 2.06  
#net-snmp|5.4.1.0|/stage/snmp/net-snmp-5.4.1|/stage/snmp/doit|Net-SNMP 5.4.1  
svn|1.4.6.1|/stage/svn/subversion-1.4.6|/stage/svn/doit|Subversion 1.4.6  
#aspell|0.60.6.0|/stage/aspell/aspell-0.60.6|/stage/aspell/doit|aspell 0.60.6  
nano|2.0.7.0|/stage/nano/nano-2.0.7|/stage/nano/doit|GNU nano 2.0.7  
#apr|1.2.12.0|/stage/apr/apr-1.2.12|/stage/apr/doit|Apache Portable Runtime 1.2.12  
#apache|2.2.8.1|/stage/apache/httpd-2.2.8|/stage/apache/doit|Apache 2.2.8  
emacs-nox|22.2.0.0|/stage/emacs/emacs-22.2|/stage/emacs/doit-nox|Emacs (no X) 22.2  
#sudo|1.6.9.16|/stage/sudo/sudo-1.6.9p16|/stage/sudo/doit|sudo 1.6.9p16
```

On the next build run, the products to be built in order are **svn**, **nano** and **emacs-nox**. Simply running:

```
build-lpp lpp-file
```

will attempt to configure, make, install, tar, package and copy to a finished location all three of the above mentioned products.

If you are having a hard go of it and the **doit** script takes some time to get right, you can also use the **-m** option of **build-lpp** to only perform a make and package and skip the **doit** altogether:

```
build-lpp -m lpp-file
```

Using **svn** as an example, two log files are produced in **/stage/log**:

```
/stage/log/svn-1.4.6.1-pkg.log  
/stage/log/svn-1.4.6.1.log
```

Even after automatic packaging, you may want to tinker with the contents. You can do this in **/stage/lpp/product/opt**. This is a good time to rename files ending in **.conf** to **.conf-dist** and update the packaging. If you are working out scripts for pre- and post-install, this is the best time to test them. Remake the package with:

```
mkinstallp -T product.template
```

## Versions

When setting a version for a package it must be a dotted quad. Again, using `svn` as an example, the package was released as 1.4.6. The initial package in `pware53` would be `pware53.svn.1.4.6.0.bff` and any patches by the maintainer would bump the last number to 1, then 2, etc.

Why would we need to bump it? New feature activated in the package at the same release level. Sometimes a poor choice in options. For example, when building `bash`, I selected an option to force POSIX always. This turned out to be a bad choice since it will never execute the `.profile` of a user. It was removed, rebuilt, packaged and released with the last number incremented by one.

The `installp` program allows for a lengthy versioning system: 12.12.1234.1234

Some packages do not allow for the a nice incremental value as `svn` did. Take `sudo` for example. A version of `sudo` is 1.6.9p16. This translates into `pware53.sudo.1.6.9.16.bff`. If an issue was discovered after release or a new option were added this might be released as `pware53.sudo.1.6.9.1601.bff`. Assuming 1.6.9p17 were released, if 1.6.9.1601 and 1.6.9.17 were in the same directory, the installer would select the 1.6.9.1601 version as it would appear newer. So maybe 1.6.9p16 should really be represented as 1.6.916.0 and retain the last digit for Pware revisions.

The last example is the model that all future release of pWare packages will follow.

## Tips

Many times a product is so thoroughly tested that it simply configures and builds right out of the box. Others need some attention to detail regarding additional libraries we would like to link into the final package. Still others will require you to dig deep into the `Makefile(s)` and `configure` scripts or the C source code to understand why it fails to make.

I have seen several places where IBM uses `-H512` and `-T512` for the linker. I cannot see how this helps improve performance or load time, but it has not produced any issues to my knowledge.

Finally here is a general set of rules that I follow so I can limit my mistakes in packaging:

Rule #0: Every package must possess the `pware.1a` licensing agreement and require the user to accept the license. I do not want to jeopardize my employment or my employer.

Rule #1: Only use `gcc` and IBM's `ld` (never GNU `ld`!).

Rule #2: Run `build-lpp` only when you have tested the `doit` and `make` process on the package. Try the process of:

```
../doit
make
```

from each package version source directory prior to attempting a package. (Obvious, I know, but sometimes things are coming up roses when along comes a riding mower with a 42" deck.)

Rule #3: When modifying source code to fix build issues, always copy an original file to filename.orig before make changes. This allows you to follow your changes and produce patches later to pass back upstream. (I have to mention it because I **still** find myself breaking this rule.)

Rule #4: When in doubt, **make distclean**. Sometimes the number of changes made to the Makefile(s), header files and C source just need a little poke.

Rule #5: Never bind to **libtermcap.a**, use **libcurses.a**.

Rule #6: Never bind to **/usr/lib/libiconv.a** unless it is completely unavoidable. This is often unavoidable when X11 is involved. Linking to both can be quite a challenge. Making the GNU **libiconv** invisible can be even more challenging especially when your support products use it.

Rule #7: Always run **ldd** against all shared objects (.so), shared object (.so) members of an archive (.a) and all executables to verify dependencies and that all references can be resolved. Some Makefiles and/or libtool like to move **/opt/pware/lib** to the **end** of an embedded **libpath**. (I am not concerned about gcc paths at the end of the list. There are no shared objects in the archive objects in the gcc available from the site.)