Title: A standard for Linux packages and version numbering

Authors: Roger Binns
Jim Winstead Jr.

Date: 24 August, 1993

Status: ALPHA

Summary

This document covers the specification for implementing packages and version numbering under Linux

Enquiries to: Roger Binns
Jim Winstead Jr.

Approvals authority:

Name	Date Approved
Roger Binns	
Jim Winstead Jr.	
HJ Lu	

Distribution list:

The Unified Linux distribution and Linux packaging mailing list

TABLE OF CONTENTS

0. DOCUMENT CONTROL

0.1. Cross references

None.

0.2. Document Predecessors

None.

0.3. Document Acceptances

None.

0.4. Changes forecast

0.4.1. Version numbering

The version numbering standard is viewed as complete. However, it will be necessary to specify when it is appropriate to use it. This will be established from when the format becomes used.

0.4.2. Packaging format

No major changes are expected to the packaging format. It is expected that a few specifics of some flags might be changed.

0.4.3. Packaging commands

Once the packaging format and commands are established, a command will be added that will allow the user to determine what package owns any particular file.

If there is sufficient demand, the package installation procedure will have an improved user interface. This will have many repercussions, as the underlying install script will still need to be usable.

The existing packaging commands may have their specifications improved so that they are less ambiguous.

0.5. Changes record

Date	Change
30 October 1992	Added future directions section. Modified map file format to use flags and types, rather than lots of different types.
29 November 1992	Cleaned it up for wider release

0.6. Change and comment procedure

For changes or comments to have an effect on this document, they must be submitted by electronic mail to all of the authors. Each change or comment must specify the section number, the increment of document referenced and a valid electronic mail return address.

Changes or comments that do not conform to the above may be ignored.

0.7. Document introduction

This document will become a standard for Linux via the procedures listed below. When it becomes a standard, all producers of packages under Linux are expected to conform to it, although it will not be enforced.

- The document will exist in alpha form until the authors are satisfied with the general content of the document. A few selected people may be invited to participate in the later stages of this.
- The document will then be released as a final alpha, and announced to the general public.
- The responses from the final alpha release will be used to produce a beta, and provide a mailing list of more general interested parties.
- Various beta releases will happen
- A freezing of changes, other than corrections will happen
- After one final beta release, a final official distribution document will be produced. At this stage, it will become a Linux standard.

0.8. Electronic Mail Addresses

This section lists the electronic mail addresses, and the period they are valid for, for all the people mentioned within the document.

Roger Binns	rogerb@x.co.uk
Jim Winstead Jr.	jwinstea@jarthur.Claremont.EDU
HJ Lu	hlu@eecs.wsu.edu

1. GENERAL

1.1. Scope

This document describes the design and implementation in the Linux environment of a standard for version numbering, and a standard for software packages.

1.2. Introduction

This document records the main specification decisions taken, provides justification for them, and provides an overview for implementation in the Linux environment. It is assumed that anyone making a package is doing so as an official release. Thus, there are no restrictions on what can be done, providing it is possible according to the specifications given in this document.

1.3. Terminology

package	A collection of files that are an installable component of software
root	A user with a numeric identifier of zero

2. VERSION NUMBERING

2.1. Description

A version numbering system is one that provides a mechanism for denoting the maturity, availability of features, and age of a product. It dictates how a progression of changes in a product will be denoted relative to the product.

2.2. Requirements

There is a big requirement for a consistent version numbering scheme under Linux. This has arisen because of the vast number of packages that make up Linux. The packages come from many vendors or authors, and usually have not been intended for Linux. This has led to Linux specific versions of packages that are disjoint from the original package. The rapid advance in Linux has also led to packages trailing behind the kernel release, which are then haphazardly updated.

The requirement is for a version numbering system that allows any Linux user to ascertain what the maturity, availability of features, and age of a product is. There are three types of version numbering that need to be active simultaneously:

- A major number giving an indication of alpha/beta/release status, that also indicates total or major rewrites
- A minor number giving an indication of the availability of new features, and the stability of those features within a major number release
- A patch number giving an indication of minor transient changes within a minor release

2.3. Objectives and constraints

The primary objective is to co-exist the three types of version numbering as listed above. As a direct result of this, any user should be able to tell from the version numbering system what a release will give him, and how important it is to him.

Because the products will usually have their own version numbering system, the Linux version numbering will be kept in parallel with the product's own version.

> Example:
>
> The Gnu compiler will have two versions, a FSF version and a Linux version. An example release would then be:
>
> GCC 2.2.2 V3L5N16

2.4. Method adopted

The method adopted is to concatenate the three components together, to produce a version string. Each component will have a set of rules as to how it changes. The version string, and associated rules will be the same for all Linux products.

2.4.1. Introduction

The version string will consist of a series of letters and digits. There will be no spaces within the string, or leading or trailing spaces as part of the string. The letters will all be in upper case, and only in the ASCII character set. The letter is specified as part of each component of the version string. The digits will be in the ASCII character set, and must be between '0' and '9' inclusive.

2.4.2. Version

This component of the version string gives an indication of alpha/beta/release status, and also indicates total or major rewrites. It would correspond to the major version number in other version numbering systems.

The character denoting this in the version string is 'V'. It is followed by one or more digits. The digits comprise an integer, which is not padded to the left with zeros.

Example:
A product with a version number of five will constitute 'V5' to the version string. A product with a version number of fifteen will constitute 'V15' to the version string.

An integer of zero will denote a product in alpha test. An integer of one will denote a product in beta test. For each major or total rewrite, or changes that render the product incompatible with previous releases, the version number will be incremented by one. The level and increment integers will then be set to zero and zero respectively.

2.4.3. Level

This component of the version string gives an indication of the availability of new features, and the stability of those features within a version release.

The character denoting this in the version string is 'L'. It is followed by one or more digits. The digits comprise an integer, which is not padded to the left with zeros.

Example:
A product with a level number of five will constitute 'L5' to the version string. A product with a level number of fifteen will constitute 'L15' to the version string.

The level numbering starts at zero, and is set to one with each new version release. When new features are available, and have been tested, and have been unchanged for two increments, except for bug fixes, they can be released. The form of the release is to increment the level integer by one.

2.4.4. Increment

This component of the version string gives an indication of the minor changes, and testing of additions within a level release. It would correspond to a patch level in other version numbering systems.

The character denoting this in the version string is 'N'. It is followed by one or more digits. The digits comprise an integer, which is not padded to the left with zeros.

Example:
A product with an increment number of five will constitute 'N5' to the version string. A product with an increment number of fifteen will constitute 'N15' to the version string.

The increment numbering starts at zero, and is set to one with each new version release. When minor changes or new features are available, but untested, they can be issued in an increment. The form of the release is to increment the level integer by one.

The rules for increments are as follows:

- All patches and other minor changes are an increment release.
- Patches and changes must increase the increment number when made to avoid any confusion whatsoever.
- For a feature to be integrated into a level, there must be an increment in which the feature is described as not having any more changes. There must be a subsequent increment with bug fixes for the feature. The following increment can constitute a new level.

2.4.5. Version string

The version string will be formed by concatenating the version, level and increments as specified above, conforming to the rules laid out in 2.4.1.

Example:
A product with a version number of 3, a level number of 7 and an increment of 50 will produce a Linux version number of 'V3L7N50'.

2.5. Application

Version strings should be applied to complete packages. They should not be applied to single files that are then grouped together. The group should have a version string in this case. If the package producer wishes, the increment can be left out of the version string, in order to reduce space required for example with filenames.

2.6. Example Product History

The table below shows the version string history of an imaginary product.

V0L0N0	V0L0N1	V0L0N2	V0L0N3	V0L1N4
V0L1N5	V0L1N6	V0L1N7	V0L2N8	V0L2N9
V0L3N10	V0L3N11	V0L3N12	V0L4N13	V0L4N14
V1L0N0	V1L0N1	V1L0N2	V1L0N3	V1L0N4
V1L1N5	V1L1N6	V1L2N7	V2L0N0	V2L0N1

3. PACKAGES

3.1. Description

A package is a collection of files, together with installation procedures that can be put on a system in order to provide a software product.

3.2. Requirements

The requirement for a package specification is to produce a description of how packages will be installed, removed, checked for accuracy of installation, and created. The specification will provide sufficient detail for programs to be created that do all of the aforementioned activities.

Any user should be easily able to create packages, as well as ascertain information about installed packages. However, only root must be able to install, or remove a package.

3.3. Objectives and constraints

Linux users have a large variety of methods of getting files onto their machines. Because of this, packages should be usable under at least all of the methods listed below:

- tape cartridges
- floppy disks
- direct ftp to the machine
- ftp to another machine, and then one of the above
- cdrom

Some of these methods will require the package to exist on a different machine, under a different operating system. Additionally, the package may be obtained on a medium that has a different capacity to that which will be used to put the package on the Linux machine. The packages must be installable from the secondary storage media listed, without it having to be copied to the hard disk first.

Each package may have parts that can be optionally installed. The individual package must be able to determine which parts to install, and which have been installed.

During package manipulation, no temporary files should be produced. This is to prevent packages consuming more than the absolute minimum of disk space.

Packages should also be easily removable. This will allow users to upgrade to newer versions properly, or not use the package or any of its files any more.

3.4. Method adopted

This section lists the various aspects of packages, and provides discussion, justification and decisions for each.

3.4.1. Existing applications

There currently exist a few applications that could be used to create packages. However, none of them meets the requirements for packaging capabilities. Among the programs considered were `tar`, `cpio` and `zoo`. These programs fail in the following requirements:

- None allow files to be easily removed (i.e. deinstall), without some extra action either by the user, or some form of shell script
- They either do not store the userid at all, or store it as an integer, making the archives non-portable between machines with different numbers for the same user
- None handle multiple parts and allow them to be placed in different locations, without actions as mentioned above
- Only cpio can handle multi-volume packages directly

3.4.2. Compression

The package will be produced in an uncompressed format. There is certainly justification for compressing it, as it will significantly reduce the time taken to distribute, and the space to store a package.

The arguments against compression are stronger. Compression requires an atomic unit to work on. The larger the unit, the better compression is obtained. There are also many compression technologies around, which change over time, and may not be available on some of the methods listed in 3.3. Thus, the package will be produced as one large atomic unit.

This unit can then compressed as the user or archive maintainers see fit. The `compress` program is recommended.

3.4.3. Atomic unit size

The package will be produced as one big file. The justification for this is that the user will view a package as a single atomic unit. It will also alleviate problems of a user not obtaining all the essential parts of a package, or different versions of each component.

3.4.4. Media size requirements

The package will not be broken up into media sizes. This is because packages can vary from the very small (e.g. 50 kilobytes) to the very large (e.g. 5 megabytes). If a package is broken into a particular media size, it may render it useless for other media sizes (e.g. 1.44MB disk is not compatible with 1.2MB disk). The package may also travel along various media before reaching the final system, and so each machine would require tools to break up, and put together the package.

It is recommended that a portable tool be used if the package is larger than the media on which it will be transported. The `cpio` program is recommended.

3.4.5. Limitations

Packages will be subject to the following limitations:

Maximum raw size	4 gigabytes

Maximum installed size	available destination media space
Minimum raw size	approx. 1 kilobyte
Minimum installed size	0 kilobytes

3.4.6. Executables provided

The following programs will be provided in order for a user to manipulate packages:

pkgadd	-	install a package/view uninstalled package
pkgrm	-	remove a package
pkgchk	-	checks accuracy of installation
pkginfo	-	provide information about an installed package

For the developer, the following program will be provided in order to produce packages:

pkgmk	-	produce a package

3.4.7. Libraries provided

A library will be provided that allows the following functions:

- utility functions, eg parsing version strings
- opening, reading and closing a map file
- opening, reading and closing an information file

The library will be called `libpackage.a`, and will be located in `/usr/lib`. A header file will also be provided, called `package.h`, and will be located in `/usr/include`.

3.4.8. File naming conventions

Each package will have its package name with '.pkg' appended to it as the file name for the package. This is to uniquely identify a package. The package name must only consist of the letters lower case 'a' to lower case 'z', and the digits '0' to '9' in the ASCII character set. They can be in any order. The longest legal package name is 8 characters in order to remain compatible with Linux file systems commonly used.

4. PROGRAM SPECIFICATIONS

4.1. pkgadd

The pkgadd program will install one or more packages from standard input. It will be able to spawn the necessary shell scripts from with the package. The program will refuse to run, unless it is running as root.

4.1.1. Internal description

The pkgadd program reads its input stream, and installs the programs as defined within it. There may be more than one package in the input stream. `pkgadd` will display the package name, important details from the information file, and if it is installed already. It will allow the user to view the package readme file before deciding if they want to install the package.

The program will extract the information file, the map file, and then the install script to the directory as specified in 5.2. The information file will then be read, and all keywords will be added to the environment, for the install script that is then invoked with the parameter '`preinstall`'.

If the return code of the installation script is zero, installation continues. If it is one, the user is queried as to whether they wish to continue. For any other values, the installation is aborted, the package directory and all files within it are deleted.

The files will then be extracted according to the map file. The parts which are extracted are determined by the PARTS keyword in the information file. If the PARTS keyword is blank, all parts are installed. Note that part 0 is always extracted.

If a pathname as specified in the map file cannot be created, or already exists, the installation is aborted. The pkgadd program must be capable of deleting all files it has installed so far.

When all files have been extracted, the information file will then be read, and all keywords will be added to the environment, for the install script which is then invoked with the parameter '`postinstall`'.

If the return code is zero, the value for the keyword STATUS is set to "Completely installed". If the return code is one, the value for the keyword STATUS is set to "Partially installed". For all other return codes, the value for the keyword STATUS is set to "Improperly installed".

The INSTDATE keyword is set to the current date and time. The FILES keyword should be modified to indicate various statistics about what has been installed. See the description of the information file for more details.

4.1.2. Command line parameters

-l non-interactive list of package details in input stream

4.2. pkgrm

The pkgrm program will remove one or more packages. It will be able to spawn the necessary install scripts from within the package. The program will refuse to run, unless it is running as root.

4.2.1. Internal description

The pkgrm program attempts to remove each package specified on its command line. The first action is to query whether the user really wants to remove the package. On receiving confirmation, all other packages are checked. If any specify this package in their USES or REQUIRES keyword, the user will be told, and given another chance to confirm removal. The information file will then be read, and all keywords will be added to the environment, for the install script that is then invoked with the parameter '`preremove`'.

If the return code is zero, the remove continues. If it is one, the user is queried if they really want to continue. If it is any other value, the remove aborts.

The files that are in the parts as specified by the PARTS keyword are then removed from the system. Any files that are 'shared' as described in the map file are left alone. If this will cause problems with certain packages, they should avoid the use of shared files. If a file is specified in the map file as being installed, but doesn't exist, the user should be notified, and queried whether they want to abort the remove, ignore the message for this file, or ignore the message for all files. If they abort, the STATUS keyword is set to "incompletely removed".

Finally, the information file is read, and all keywords will be added to the environment, for the install script that is then invoked with the parameter '`postremove`'.

The return code of pkgrm is then the return code of the execution. The package directory and its administrative files are deleted first.

4.2.2. Command line parameters

<none> help information is displayed

4.3. pkginfo

This program obtains information about a package. It will refuse to run if the -w parameter is given, and it is not running as root.

4.3.1. Internal description

There are various actions the pkginfo program will perform on one or more packages. The first is displaying information. For this, all the keywords that are mandatory in a package information file, as well as some other useful ones, and their values are printed. If the verbose flag (-v) was given, then all keywords that contain upper case letters, and their associated values are given. If a package couldn't be found, then the return code is one, else it is zero.

The second action is displaying a keyword from an information file. In this instance, the value associated with the keyword is printed. If it cannot be found, nothing is printed. The return code is zero if the package and keyword were found, one if the package but not the keyword were found, and two for all other situations. This mode is intended for installation scripts to find out information about other packages on the system.

The third action is to write to a keyword in an information file. If the package cannot be found, the return code is 1, and nothing further happens. If the package is found, the value is changed if it already exists, or added if it doesn't. The return code is zero.

4.3.2. Command line parameters

The three forms have the following parameters:

information:	[-v] package keyword:
read keyword:	-r package keyword
write keyword:	-w package keyword value

4.4. pkgchk

This program will check the accuracy of an installation. This will include checking all necessary files are present, and have the correct attributes.

4.4.1. Internal description

The program will read through the map file for the package, and for all the parts present in the PARTS keyword, will check each of the attributes for the file is correct, unless marked as changeable.

If there is nothing wrong with the installation, the return code will be zero. If the package is not found, it will be 1. For all other cases, it will be 2.

4.4.2. Command line parameters

The only parameter is a single package name.

4.5. pkgmk

This program will produce a package on its output stream. The current user and group id must be able to read all files required to produce the package. It must be invoked from a directory containing the pkginfo and pkgproto files for the package.

4.5.1. Internal description

The pkgmk program operates in two stages. The first stage is to read in the pkginfo file. All required keywords should be present. If the following keywords are missing, they will be added. If they already exist, the values will be changed to those indicated.

PSTAMP	hostnameYYMMDDHHMMSS
INSTDATE	Jan 01 1990 00:00
STATUS	Uninstalled
FILES	0 0 0 0 0

The program will then build up the map file using the prototype file. Internally, it will build a list of files that will be streamed to produce the package output stream. After producing valid pkgmap and pkginfo files, pkgmk will then produce the package streamed to standard output when the program was invoked.

5. DIRECTORIES

5.1. Package programs

The programs provided for package manipulation will be placed in `/usr/bin`. It will consist of one file called pkg, with the following symbolic links: pkgmk, pkginfo, pkgadd, pkgrm, pkgchk.

5.2. Package configuration

The following parts of a package will be kept in `/usr/spool/pkg/<package name>`, and will have names as given.

Information file	info
Map file	map
Install script	install

Additionally, a package installation script may save files such as configuration files in `/etc/save/<package name>`. It could do this for example during preremove. On the next postinstall, the files can be restored.

5.3. Package files

The files produced by installing the package will be placed in the directories as specified in the map file, or if the files are relocatable in the location specified by the user. No other copies are kept.

6. FILE FORMATS

6.1. Package file

6.1.1. Overview

A package will be a concatenation of the following parts:

- an identifying header
- the information file
- the readme file
- the map file
- the installation script
- each file as specified in the map

6.1.2. Identifying header

The header will consist of a sequence of bytes as follows:

Bytes	Content
0 to 7	'LinuxPkg'
8 to 17	The version string of pkgmk that produced the program, padded to the right with spaces
18 to 25	The package name according to the naming conventions specified earlier, padded to the right with spaces

6.1.3. Information file

The contents of the information file are specified elsewhere. Within the package, it is stored as a sequence of bytes denoting the length in ASCII, a NULL character, and then a byte stream of the file.

6.1.4. Readme file

This file can be displayed by the user prior to making the decision to install. It should contain a very brief description of the package, together with a few major points, and an indication of space requirements.

Within the package, it is stored as a sequence of bytes denoting the length in ASCII, a NULL character, and then a byte stream of the file.

6.1.5. Map file

The contents of the map file are specified elsewhere. Within the package, it is stored as a sequence of bytes denoting the length in ASCII, a NULL character, and then a byte stream of the file.

6.1.6. Installation Script

This script is invoked by the pkgadd and pkgrm commands. Within the package, it is stored as a sequence of bytes denoting the length in ASCII, a NULL character, and then a byte stream of the file.

6.1.7. Package files

The files in the rest of the package are in the rest of the raw package. The files are in order as specified in the map file. Only files with physical data are present. This means that the following types are not "d,x,l,p,c,b,s". Each file is stored as a sequence of bytes denoting the length in ASCII, a NULL character, and then a byte stream of the file.

6.2. Information file

The information file is used to specify information about the product. It is in the form:

keyword = "value"

The restrictions on the keyword is that it is at least 1 character long, and no more than 50 characters long. The minimum length for a value is 0 characters, and the maximum length is 1024 characters.

The following keywords are specified. Case is significant. Any keyword with a * next to it is mandatory.

Keyword	Description
PKGINST*	package name
NAME*	free form text package name
CATEGORY*	space delimited list of categories the package belongs to
ARCH*	the architecture for which the binaries were compiled (e.g. i386, i486)
VERSION*	major version number
LEVEL*	level number
INCREMENT*	increment number of package
REQUIRES	a list of packages required for installation - this can be used to abort the installation prior to reading the map file. It will also ensure that the user receives a warning if they try to pkgrm the packages listed. The package names are separated by white space.
USES	a list of packages that it is desirable to have for installation. The user will be notified of this if any listed are not present on the system. It will also ensure that the user receives a warning if they try to pkgrm the packages listed. The package names are separated by whitespace.
PRODUCER	producer of package
SUPPORT	email address for package support
PSTAMP*	unique stamp for package producer to identify package
INSTDATE*	date and time package installed
STATUS*	the installation status of the package

PARTS*	a space delimited list of parts installed
FILES*	a list of integers with details about the installed package, each separated by white space. The integers are number of installed pathnames, number of shared pathnames, number of directories, number of executables, number of kilobytes used

In addition, a package may add its own keywords. These can be manipulated using the pkginfo command. If you do not wish a keyword to be visible to users, even with a -v switch, make it entirely lower case.

6.3. Map file

The map file is the main heart of a package. It gives details of each file in a package, and where and how it should be installed. Lines starting with a hash sign '#' are comments. Note that any part of the line can contain variables. These are as specified in the package information file, and are specified as a dollar sign, an open round bracket, a sequence of digits and letters, and close round bracket. Case is significant. The variable is expanded after the line is parsed, so it cannot contain more than one component.

Each line starts with three components, separated with colons.

part ASCII digits denoting the part number of the file

type a set of letters denoting the file type, and optional flags

type	flags	meaning
<any>	i	only read this line during package installation
	r	only read this line during package removal
f		a file
	S	shared - the file is only installed if the existing file is older. This flag also implies the 'i' flag, as it will not be removed
d		a directory
	x	exclusive - the directory should contain ONLY files listed in the package map file
l		link
	h	the link is a hard link (i.e. the default is soft)
D		device
	b	the device is a block device (i.e. the default is character)
p		named pipe
R		required - gives a package name and performs a comparison on its version string.

pathname the name of the file
if the type is link, this field will be in the form of pathname=pathname
if the type is required, this field will be a package name

The rest of the line is dependent on the file type. Below is a list of components of the rest of the line, which are separated by a colon.

Component	Meaning
major	major device number

minor	minor device number
mode	file permissions in octal
owner	user id of file owner (in text)
group	group id of file owner (in text)
size	the size of the file in bytes
cksum	a checksum of the file
modtime	the last modification time of the file, in seconds since 1970

The following table lists the components used in each file type.

	Major	Minor	Mode	Owner	Group	Size	Cksum	Modtime
f,l			x	x	x	x	x	x
p,d			x	x	x			
D	x	x	x	x	x			

However, for the required type 'R', the remaining fields are an operator for comparison, and a version string. The allowed boolean specifiers are >, <, =, >=, <=, !. If the operator is not (!), the two packages cannot co-exist. The version string should follow the standard laid out earlier in this document.

If a component can change in time, the component can have a 'v' as the first character. If a group or owner need an id starting with a 'v', the first character can be a space. The component must always be present. Doing this will make pkgchk ignore the changed file attribute.

6.4. Prototype file

The prototype file is what the package producer uses to specify what files go into the package, and what their entries will be in the map file.

It takes the following form. Any line starting with a comment or is blank is copied verbatim. a line starts with anything except a full stop, the line is copied to the map file. Any fields with a question mark in them will have it replaced with the value from an existing pathname. That means that if the pathname has variable expansion, the variable must be present in the info file, and the field must resolve to an existing pathname. It can be installed elsewhere by changing the value during the preinstall phase.

If the line starts with a full stop, the next line must contain a file type. For each file that matches the following file specification, a line is produced in the map file, with the $F replaced by the last component of the file specification wildcard.

The following is a two line example:

```
. src/xeyes/*.c
1:f:$(srcdir)/xeyes/$F:?:$(srcuid):$(srcgrp):?:?:?
```

The special token '$F' expands to the last component of the file as specified in the line containing the wildcard. The map file produced will then contain the following for those two lines:

```
1:f:$(srcdir)/xeyes/main.c:755:$(srcuid):$(srcgrp):123:456:789
1:f:$(srcdir)/xeyes/graphics.c:755:$(srcuid):$(srcgrp):012:345:678
1:f:$(srcdir)/xeyes/server.c:755:$(srcuid):$(srcgrp):912:345:678
```

7. EXAMPLE

This section shows an example package, and how it would be created. Also given are generic examples of installing from some of the media supported.

7.1. Package description

The package being simulated is a release of X-windows. It is in no way representative of the real X-windows, but is used because it is familiar.

The package contains the following parts:

- essential binaries, one set i386, one set i486 (compulsory)
- fonts (75 dpi)
- fonts (120 dpi) (must have one set of fonts)
- demonstration clients (optional)
- source for demonstration clients (optional)
- other useful binaries (recommended)

The first step is to create a package information file:

```
PKGINST="xwindows"
NAME="X-Windows for Linux"
CATEGORY="system graphics"
ARCH="i386"
VERSION="1"
LEVEL="3"
INCREMENT="15"
SUPPORT="Xwin@support.xwin.bbs.edu"
PARTS=""
```

Having created an information file, a prototype file is needed. The file will look like the following:

```
# x windows for Linux
# package made 1992 05 27 on pandora@company.com

# part 0 is required packages
.
0:R:kernel:>=:V3L2N10

# part 1 is essential binaries i386
. src/bin/i386/essential/.*
1:f:/usr/bin/X11/$F:1755:root:wheel:?:?:?

#part 2 is essential binaries i486
. src/bin/i486/essential/.*
2:f:/usr/bin/X11/$F:1755:root:wheel:?:?:?

# part 3 is 75 dpi fonts
3:d:/usr/lib/X11/fonts/75dpi:755:bin:bin
. fonts/75dpi/.*
3:f:/usr/lib/X11/fonts/75dpi/$F:?:?:?:?:?

# part 4 is 120 dpi fonts
4:d:/usr/lib/X11/fonts/120dpi:755:bin:bin
. fonts/120dpi/.*
```

```
4:f:/usr/lib/X11/fonts/120dpi/$F:?:?:?:?:?

# part 5 is demonstration binaries
5:d:$(clientbin):755:bin:bin
. src/demos/bin/.*
5:f:$(clientbin)/$F:755:bin:bin:?:?:?

# part 6 is demonstration source
6:d:$(clientsrc):755:$srcuid:$srcgrp
. src/demos/xeyes/.*\.[ch]
6:f:$(clientsrc)/xeyes/$F:$srcuid:$srcgrp:?:?:?
. src/demos/xmaze/.*\.[ch]
6:f:$(clientsrc)/xmaze/$F:$srcuid:$srcgrp:?:?:?

# part 7 is other useful binaries
. src/bin/others/.*
7:f:/usr/bin/X11/$F:755:bin:bin:?:?:?
```

`pkgmk` will produce a map file like the following from the above prototype file. It is only shown as an example, and the package producer will not normally have worry about map files.

```
# x windows for Linux
# package made 1992 05 27 on pandora@company.com

# part 0 is required packages
0:R:kernel:>=:V1L3N15

# part 1 is essential binaries i386
1:f:/usr/bin/X11/startx:1755:root:wheel:76587:786876:787
1:f:/usr/bin/X11/xinit:1755:root:wheel:786:78687:763
1:f:/usr/bin/X11/X386:1755:root:wheel:675:67567:76576

# part 2 is essential binaries i486
2:f:/usr/bin/X11/startx:1755:root:wheel:76587:786876:787
2:f:/usr/bin/X11/xinit:1755:root:wheel:786:78687:763
2:f:/usr/bin/X11/X386:1755:root:wheel:675:67567:76576

# part 3 is 75dpi fonts
3:d:/usr/lib/X11/fonts/75dpi
3:f:/usr/lib/X11/fonts/75dpi/courier.fon:555:bin:bin:987436:54334:435

# part 4 is 120dpi fonts
4:d:/usr/lib/X11/fonts/120dpi
4:f;/usr/lib/X11/fonts/120dpi/courier.fon:555:bin:bin:987436:54334:435

# part 5 is demonstration binaries
5:d:$(clientbin):755:bin:bin
5:f:$(clientbin)/xeyes:755:bin:bin:57684:12356754:56546
5:f:$(clientbin)/xclock:755:bin:bin:7678678:786786:786876
5:f:$(clientbin)/xmaze:755:bin:bin:657:756:546

# part 6 is source for demonstration binaries
6:d:$(clientsrc):755:$(srcuid):$(srcgrp)
6:dx:$(clientsrc)/xeyes:755:$(srcuid):$(srcgrp)
6:f:$(clientsrc)/xeyes/xeyes.h:755:$(srcuid):$(srcgrp):7856:576:4378
6:f:$(clientsrc)/xeyes/xeyes.c:755:$(srcuid):$(srcgrp):879456:54897:5497
6:dx:$(clientsrc)/xmaze:755:$(srcuid):$(srcgrp)
6:f:$(clientsrc)/xmaze/xmaze.h:755:$(srcuid):$(srcgrp):7856:576:4378
6:f:$(clientsrc)/xmaze/xmaze.c:755:$(srcuid):$(srcgrp):879456:54897:5497

# part 7 is other useful binaries
7:f:/usr/bin/X11/xhost:755:bin:bin:897:87897:3432
```

```
7:f:/usr/bin/X11/xsetroot:755:bin:bin:8947:89798:9898
```

After creating the map file, an installation script is needed. This could be done in a C program, but I have chosen to implement it as a Bourne shell script.

```
#! /bin/sh

case $1 in
   preinstall)
      # code to ask which parts to install
      echo "Do you want to use binaries compiled for the 486? [YyNn]"
      read answer
      if [$answer == "y" -o $answer == "Y"];  then
         pkginfo -w $PKGINST ARCH i486
         pkginfo -w $PKGINST PARTS 1
      else
         pkginfo -w $PKGINST PARTS 2
      fi
      echo "Do you wish to use 75 dpi fonts? [YyNn]"
      echo "75 dpi fonts are recommended on 14 inch monitors"
      echo "up to 800x600.  Otherwise 120 dpi fonts will be used."
      read answer
      if [$answer == "y" -o $answer == "Y"]; then
         pkginfo -w $PKGINST PARTS "`pkginfo -r $PKGINST PARTS`  3"
      else
         pkginfo -w $PKGINST PARTS "`pkginfo -r $PKGINST PARTS`  4"
      fi
      echo "Do you want some demonstration clients? [YyNn]"
      read answer
      if [$answer == "y" -o $answer == "Y"]; then
         pkginfo -w $PKGINST PARTS "`pkginfo -r $PKGINST PARTS` 5"
         echo "Where do you want them installed (suggest /usr/local/bin)?"
         read answer
         pkginfo -w $PKGINST clientbin "$answer"
         echo "Do you want the source for the clients? [YyNn]"
         read answer
         if [$answer == "y" -o $answer == "Y"]; then
            pkginfo -w $PKGINST PARTS "`pkginfo -r $PKGINST PARTS` 6"
            echo "Where do you want them installed (suggest /usr/src/X11/demos)?"
            read answer
            pkginfo -w $PKGINST clientsrc "$answer"
            echo "What userid do you want the client source to have (suggest src)?"
            read answer
            pkginfo -w $PKGINST srcuid "$answer"
            echo "What group id do you want the client source to have (suggest src)?"
            read answer
            pkginfo -w $PKGINST srcgrp "$answer"
         fi
      fi
      echo "Do you want other clients (recomended)? [YyNn]"
      read answer
      if [$answer == "y" -o $answer == "Y"]; then
         pkginfo -w $PKGINST PARTS "`pkginfo -r $PKGINST PARTS` 7"
      fi
      exit 0 ;;
   postinstall)
      exit 0 ;;
   preremove)
      exit 0 ;;
   postremove)
      exit 0;;
esac
```

The final step is to run pkgmk. This needs to be in the same directory as the `info`, `readme` and `proto` files.

```
linux% pkgmk | compress >xwindows.pkg.Z
```

This final product can then be distributed as needed.

7.2. Installation from ftp site

7.2.1. Single package

This example shows how a package could be installed from an ftp site. The package is assumed to be larger than the media available.

```
ftp> get gcc.pkg.Z
...
unix% cat gcc.pkg.Z | cpio ??? /dev/floppy0 -
...
linux% cpio ??? /dev/fd0 - | uncompress | pkgadd
```

7.2.2. Multiple packages

This example shows how multiple packages could be installed from an ftp site. The packages are assumed to be in a distribution of sizes from less than the media capacity to more.

```
ftp> mget *.pkg.Z
...
unix% zcat *.pkg.Z | compress | cpio ??? /dev/floppy0 -
...
linux% cpio ??? /dev/fd0 - | uncompress | pkgadd
```

7.2.3. Direct onto Linux machine with ftp

This example shows a package being installed from an ftp site directly onto the local Linux machine. It requires a named pipe, denoted here by the filename '`named.pipe`'.

```
ftp> get package.pkg.Z named.pipe
```

on another virtual terminal:

```
linux% zcat named.pipe | pkgadd
```

7.3. Multiple disks

This subsection shows how a multiple disk, multiple package installation would be done. Current examples of this are MCC-interim and SLS.

```
linux% cpio ??? /dev/fd0 - | uncompress | pkgadd
```