Packaging Case Studies B

Introduction

This appendix presents case studies to show packaging techniques such as installing objects conditionally, determining at run time how many files to create, and modifying an existing data file during package installation and removal.

Each case begins with a description of the study, followed by a list of the packaging techniques used and a narrative description of the approach taken when using those techniques. After this material, sample files and scripts associated with the case study are shown.

Case #1: Using a `request` *Script*

This package has three types of objects. The installer may choose which of the three types to install and where to locate the objects on the installation machine.

Techniques

This case study shows examples of the following techniques:

- Using variables in object pathnames
- Using the request script to solicit input from the installer
- Setting conditional values for an installation parameter

Approach

To set up selective installation, you must:

- Define a class for each type of object that can be installed.

  In this case study, the three object types are the package executables, the manual pages, and the `emacs` executables. Each type has its own class: `bin`, `man`, and `emacs`, respectively. Notice that in the `prototype` file all the object files belong to one of these three classes.

- Initialize the `CLASSES` parameter in the `pkginfo` file to all classes.

  Normally when you define a class, you should list that class in the `CLASSES` parameter. Otherwise, no objects in that class are installed. For this example, the parameter is initially set to all classes. Since the `pkginfo` file contains the default settings, all classes are included. `CLASSES` is given values by the request script, based on the package pieces chosen by the installer. This way, `CLASSES` is set to only those object types that the installer wants installed. The first figure shows the `pkginfo` file associated with this package. Notice that the `CLASSES` parameter is set to null.

- Use variables to define object pathnames in the `prototype` file.

  The request script sets these variables to the value which the installer provides. `pkgadd` resolves these variables at installation time and so knows where to install the package.

  The three variables used in this example are set to their default in the pkginfo file and serve the following purposes:
  - `$NCMPBIN` defines the location for object executables
  - `$NCMPMAN` defines the location for manual pages
  - `$EMACS` defines the location for `emacs` executables

  The example prototype file shows how to define the object pathnames with variables.

- Create a request script to ask the installer which parts of the package should be installed and where they should be placed.

  The request script for this package asks two questions:
  - Should this part of the package be installed?

When the answer is yes, the appropriate class name is added to the `CLASSES` parameter. For example, when the installer chooses to install the manual pages associated with this package, the class `man` is added to the `CLASSES` parameter.

- If so, where should that part of the package be placed?

  The appropriate variable is set to the response to this question. In the manual page example, the variable `$NCMPMAN` is set to the response value.

  These two questions are repeated for each of the three object types.

At the end of the request script, the parameters are made available to the installation environment for `pkgadd` and any other packaging scripts. The request script does this by writing these definitions to the file provided by the calling utility. For this example, no other scripts are provided.

When looking at the request script for this example, notice that the questions are generated by the data validation tools `ckyorn` and `ckpath`. See also `ckyorn`(1) and `ckpath`(1).

Sample Files

The following example shows the `pkginfo` file for Case #1:

```
Pkg=ncmp
NAME=NCMP Utilities
CATEGORY=application, tools
BASEDIR=/
ARCH=SPARC
VERSION=RELEASE 1.0, Issue 1.0
CLASSES=bin emacs man
NCMPBIN=/bin
NCMPMAN=/usr/man
EMACS=/usr/emacs
```

B

This example file shows the `prototype` file for Case #1:

```
i pkginfo
i request
x bin $NCMPBIN 0755 root other
f bin $NCMPBIN/dired=/usr/ncmp/bin/dired 0755 root other
f bin $NCMPBIN/less=/usr/ncmp/bin/less 0755 root other
f bin $NCMPBIN/ttype=/usr/ncmp/bin/ttype 0755 root other
f emacs $NCMPBIN/emacs=/usr/ncmp/bin/emacs 0755 root other
x emacs $EMACS 0755 root other
f emacs $EMACS/ansii=/usr/ncmp/lib/emacs/macros/ansii 0644 root
other
f emacs $EMACS/box=/usr/ncmp/lib/emacs/macros/box 0644 root
other
f emacs $EMACS/crypt=/usr/ncmp/lib/emacs/macros/crypt 0644 root
other
f emacs $EMACS/draw=/usr/ncmp/lib/emacs/macros/draw 0644 root
other
f emacs $EMACS/mail=/usr/ncmp/lib/emacs/macros/mail 0644 root
other
f emacs $NCMPMAN/man1/emacs.1=/usr/ncmp/man/man1/emacs.1 0644
root other
d man $NCMPMAN 0755 root other
d man $NCMPMAN/man1 0755 root other
f man $NCMPMAN/man1/dired.1=/usr/ncmp/man/man1/dired.1 0644 root
other
f man $NCMPMAN/man1/ttype.1=/usr/ncmp/man/man1/ttype.1 0644 root
other
f man $NCMPMAN/man1/less.1=/usr/ncmp/man/man1/less.1 0644 inixmr
other
```

This example file shows the request script for Case #1:

```
trap 'exit 3' 15
# determine if and where general executables should be placed
ans=`ckyorn -d y \
-p "Should executables included in this package be installed"
` || exit $?
if [ "$ans" = y ]
then
CLASSES="$CLASSES bin"
NCMPBIN=`ckpath -d /usr/ncmp/bin -aoy \
-p "Where should executables be installed"
` || exit $?
fi
# determine if emacs editor should be installed, and if it should
# where should the associated macros be placed
ans=`ckyorn -d y \
-p "Should emacs editor included in this package be installed"
` || exit $?
if [ "$ans" = y ]
then
CLASSES="$CLASSES emacs"
EMACS=`ckpath -d /usr/ncmp/lib/emacs -aoy \
-p "Where should emacs macros be installed"
` || exit $?
```

Note that the request script can exit without leaving any files on the filesystem. For installations on Solaris versions prior to 2.5 (where no `checkinstall` script may be used) the request script is the correct place to test the file system in any manner necessary to assure that the installation will succeed. When the request script exits with code 1, the installation will quit cleanly.

These examples show the use of parametric paths to establish multiple base directories. It is necessary to show how this is done, but the preferred method involves use of the `$BASEDIR` parameter which is managed and validated by `pkgadd`. Whenever multiple base directories are used, special care must be taken to provide for installation of multiple versions and architectures on the same platform.

B

Case #2: Using Classes, Class Action Scripts, and the `space` *File*

This study creates a database file at installation and saves a copy of the database when the package is removed.

Techniques

This case study shows examples of the following techniques:

- Using classes and class action scripts to perform special actions on different sets of objects
- Using the `space` file to inform `pkgadd` that extra space is required to install this package properly
- Using the `installf` command

Approach

To create a database file at installation and save a copy on removal, you must:

- Create three classes.

  This package requires the following three classes to be defined in the `CLASSES` parameter:
  - The standard class of `none` (contains a set of processes belonging in the subdirectory `bin`)
  - The `admin` class (contains an executable file `config` and a directory containing data files)
  - The `cfgdata` class (contains a directory)
- Make the package collectively relocatable.

  Notice in the `prototype` file that none of the pathnames begins with a slash or a variable. This indicates that they are collectively relocatable.
- Calculate the amount of space the database file requires and create a `space` file to deliver with the package. This file notifies `pkgadd` that the package requires extra space and specifies how much extra space.
- Create an installation class action script for the `admin` class.

  The script shown initializes a database using the data files belonging to the `admin` class. To perform this task, it:
  - Copies the source data file to its proper destination

- Creates an empty file named `config.data` and assigns it to a class of `cfgdata`
- Executes the `bin/config` command (delivered with the package and already installed) to populate the database file `config.data` using the data files belonging to the `admin` class
- Executes `installf -f` to finalize installation

  No special action is required for the `admin` class at removal time so no removal class action script is created. This means that all files and directories in the `admin` class are removed from the system.

- Create a removal class action script for the `cfgdata` class.

  The removal script makes a copy of the database file before it is deleted. No special action is required for this class at installation time, so no installation class action script is needed.

  Remember that the input to a removal script is a list of path names to remove. Path names always appear in lexical order with the directories appearing first. This script captures directory names so that they can be acted upon later and copies files to a directory named `/tmp`. When all the path names have been processed, the script then goes back and removes all directories and files associated with the `cfgdata` class.

  The outcome of this removal script is to copy `config.data` to `/tmp` and then remove the `config.data` file and the data directory.

Sample Files

The following example file shows the `pkginfo` file for Case #2:

```
PKG=krazy
NAME=KrAzY Applications
CATEGORY=applications
BASEDIR=/
ARCH=SPARC
VERSION=Version 1
CLASSES=none cfgdata admin
```

This example file shows the prototype file for Case #2:

```
i pkginfo
i request
i i.admin
i r.cfgdata
d none bin 555 root sys
f none bin/process1 555 root other
f none bin/process2 555 root other
f none bin/process3 555 root other
f admin bin/config 500 root sys
d admin cfg 555 root sys
f admin cfg/datafile1 444 root sys
f admin cfg/datafile2 444 root sys
f admin cfg/datafile3 444 root sys
f admin cfg/datafile4 444 root sys
d cfgdata data 555 root sys
```

The following example file shows the `space` file:

```
# extra space required by config data which is
# dynamically loaded onto the system
data 500 1
```

The following example file shows the installation script:

```
# PKGINST parameter provided by installation service
# BASEDIR parameter provided by installation service
while read src dest
do
# the installation service provides '/dev/null' as the
# pathname for directories, pipes, special devices, etc.
# which it knows how to create.
[ "$src" = /dev/null ] && continue
cp $src $dest || exit 2
done
# if this is the last time this script will be executed
# during the installation, do additional processing here.
if [ "$1" = ENDOFCLASS ]
then
# our config process will create a data file based on any changes
# made by installing files in this class; make sure the data file
# is in class 'cfgdata' so special rules can apply to it during
# package removal.
installf -c cfgdata $PKGINST $BASEDIR/data/config.data f 444 root
sys || exit 2
$BASEDIR/bin/config > $BASEDIR/data/config.data || exit 2
installf -f -c cfgdata $PKGINST || exit 2
fi
exit 0
```

This illustrates a rare instance in which `installf` is appropriate in a class action script. Since a space file has been used to reserve room on a specific filesystem, this new file may be safely added even though it is not included in the pkg map.

Normal `installf`'s should be saved for use in the postinstall script after the package database is stable.

This example file shows the removal script for Case #2:

```
# the product manager for this package has suggested that
# the configuration data is so valuable that it should be
# backed up to /tmp before it is removed!
while read path
do
# pathnames appear in lexical order, thus directories
# will appear first; you can't operate on directories
# until done, so just keep track of names until
# later
if [ -d $path ]
then
dirlist="$dirlist $path"
continue
fi
mv $path /tmp || exit 2
done
if [ -n "$dirlist" ]
then
rm -rf $dirlist || exit 2
fi
exit 0
```

Case #3: Using `copyright`, `compver`, *and* `depend` *Files*

This package uses the optional packaging files to define package compatibilities and dependencies and to present a copyright message during installation.

Techniques

This case study shows examples of the following techniques:

- Using the `copyright` file
- Using the `compver` file
- Using the `depend` file

Approach

To meet the requirements in the description, you must:

- Create a `copyright` file.

  A `copyright` file contains the ASCII text of a copyright message. The message shown in the figure is displayed on the screen during package installation (and also during package removal).

- Create a `compver` file.

  The `pkginfo` file shown in the next figure defines this package version as version 3.0. The `compver` file defines version 3.0 as being compatible with versions 2.3, 2.2, 2.1, 2.1.1, 2.1.3 and 1.7.

- Create a `depend` file.

  Files listed in a `depend` file must already be installed on the system when a package is installed. The example shown has 11 packages which must already be on the system at installation time.

Sample Files

The following example file shows the `pkginfo` file for Case #3:

```
PKG=case3
NAME=Case Study #3
CATEGORY=application
BASEDIR=/
ARCH=SPARC
VERSION=Version 3.0
CLASSES=none
```

This example file shows the `copyright` file for Case #3:

```
Copyright (c) 1989 company_name
All Rights Reserved.
THIS PACKAGE CONTAINS UNPUBLISHED PROPRIETARY SOURCE CODE OF
company_name.
The copyright notice above does not evidence any
actual or intended publication of such source code
```

The following example file shows the `compver` file for Case #3:

```
Version 2.3
Version 2.2
Version 2.1
Version 2.1.1
Version 2.1.3
Version 1.7
```

This example file shows the `depend` file for Case #3:

```
P acu Advanced C Utilities
Issue 4 Version 1
P cc C Programming Language
Issue 4 Version 1
P dfm Directory and File Management Utilities
P ed Editing Utilities
P esg Extended Software Generation Utilities
Issue 4 Version 1
P graph Graphics Utilities
P rfs Remote File Sharing Utilities
Issue 1 Version 1
P rx Remote Execution Utilities
P sgs Software Generation Utilities
Issue 4 Version 1
P shell Shell Programming Utilities
P sys System Header Files
Release 3.1
```

Case #4a: Using the `sed` *Class and a* `postinstall` *Script*

This study modifies a file which exists on the installation machine during package installation. It uses one of three modification methods. The other two methods are shown in Cases #4b and #4c. The file modified is `/sbin/inittab`.

Techniques

This case study shows examples of the following techniques:

- Using the `sed` class

- Using a `postinstall` script

Approach

To modify `/sbin/inittab` at the time of installation, you must:

- Add the `sed` class script to the `prototype` file.

  The name of a script must be the name of the file that will be edited. In this case, the file to be edited is `/sbin/inittab` and so our sed script is named `/sbin/inittab`. There are no requirements for the `mode owner group` of a sed script (represented in the sample `prototype` by question marks). The file type of the `sed` script must be `e` (indicating that it is editable).

- Set the `CLASSES` parameter to include `sed`.

  In the case of the example shown next, `sed` is the only class being installed. However, it could be one of any number of classes.

- Create a `sed` class action script.

  You cannot deliver a copy of `/sbin/inittab` that looks the way you need it to, since `/sbin/inittab` is a dynamic file and you have no way of knowing how it will look at the time of package installation. Using a `sed` script allows us to modify the `/sbin/inittab` file during package installation.

  As already mentioned, the name of a `sed` script should be the same as the name of the file it will edit. A `sed` script contains `sed` commands to remove and add information to the file.

- Create a `postinstall` script.

  You need to execute `init q` to inform the system that `/sbin/inittab` has been modified. The only place you can perform that action in this example is in a `postinstall` script. Looking at the example `postinstall` script, you will see that its only purpose is to execute the following command:

```
init q
```

  This approach to editing `/sbin/inittab` during installation has two drawbacks. First of all, you have to deliver a full script (the `postinstall` script) simply to perform `init q`. In addition to that, the package name at the end of each comment line is hardcoded.

Sample Files

This example file shows the `pkginfo` file for Case #4a:

```
KG=case4a
NAME=Case Study #4a
CATEGORY=applications
BASEDIR=/
ARCH=SPARC
VERSION=Version 1d05
CLASSES=sed
```

The following example file shows the `prototype` file for Case #4a:

```
i pkginfo
i postinstall
e sed /sbin/inittab ? ? ?
```

This example file shows the `sed` class action script for Case #4a:

```
!remove
# remove all entries from the table that are associated
# with this package, though not necessarily just
# with this package instance
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d
!install
# remove any previous entry added to the table
# for this particular change
/^[^:]*:[^:]*:[^:]*:[^#]*#ROBOT$/d
# add the needed entry at the end of the table;
# sed(1) does not properly interpret the '$a'
# construct if you previously deleted the last
# line, so the command
# $a\
# rb:023456:wait:/usr/robot/bin/setup #ROBOT
# will not work here if the file already contained
# the modification. Instead, you will settle for
# inserting the entry before the last line!
$i\
rb:023456:wait:/usr/robot/bin/setup #ROBOT
```

This example file shows the `postinstall` script for Case #4a:

```
# make init re-read inittab
/sbin/init q ||
exit 2
exit 0
```

Case #4b: Using Classes and Class Action Scripts

This study modifies an existing file during package installation. It uses one of three modification methods. The other two methods are shown in Cases #4a and #4c. The file modified is `/sbin/inittab`.

Techniques

This case study shows examples of the following techniques:

- Creating classes
- Using installation and removal class action scripts

Approach

To modify `/sbin/inittab` during installation, you must:

- Create a class.

  Create a class called `inittab`. You must provide an installation and a removal class action script for this class. Define the `inittab` class in the `CLASSES` parameter in the `pkginfo` file.

- Create an `inittab` file.

  This file contains the information for the entry that you will add to `/sbin/inittab`. Notice in the `prototype` file figure that `inittab` is a member of the `inittab` class and has a file type of `e` for editable.

- Create an installation class action script.

  Since class action scripts must be multiply executable (meaning you get the same results each time they are executed), you cannot just add the sample text to the end of the file. The class action script performs the following procedures:
  - Checks to see if this entry has been added before
  - If it has, removes any previous versions of the entry
  - Edits the `inittab` file and adds the comment lines so you know where the entry is from
  - Moves the temporary file back into `/sbin/inittab`
  - Executes `init q` when it receives the end-of-class indicator

  Note that `init q` can be performed by this installation script. A one-line `postinstall` script is not needed by this approach.

- Create a removal class action script.

  The `removal` script is very similar to the installation script. The information added by the installation script is removed and `init q` is executed.

This case study resolves the drawbacks to Case #4a. You can support multiple package instances since the comment at the end of the `inittab` entry is now based on package instance. Also, you no longer need a one-line `postinstall` script. However, this case has a drawback of its own. You must deliver two class action scripts and the `inittab` file to add one line to a file. Case #4c shows a more streamlined approach to editing `/sbin/inittab` during installation.

Sample Files

This example file shows the `pkginfo` file for Case #4b:

```
PKG=case4b
NAME=Case Study #4b
CATEGORY=applications
BASEDIR=/
ARCH=SPARC
VERSION=Version 1d05
CLASSES=inittab
```

The following example file shows the `prototype` file for Case #4b:

```
i pkginfo
i i.inittab
i r.inittab
e inittab /sbin/inittab ? ? ?
```

This example file shows the installation class action script for Case #4b:

```
# PKGINST parameter provided by installation service
while read src dest
do
# remove all entries from the table that
# associated with this PKGINST
sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" $dest >
/tmp/$$itab ||
exit 2
sed -e "s/$/#$PKGINST" $src >> /tmp/$$itab ||
exit 2
mv /tmp/$$itab $dest ||
exit 2
done
if [ "$1" = ENDOFCLASS ]
then
/sbin/init q ||
exit 2
fi
exit 0
```

The following example file shows the `removal` script for Case #4b:

```
# PKGINST parameter provided by installation service
while read src dest
do
# remove all entries from the table that
# are associated with this PKGINST
sed –e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" $dest >
/tmp/$$itab ||
exit 2
mv /tmp/$$itab $dest ||
exit 2
done
/sbin/init q ||
exit 2
exit 0
```

This example file shows the one line `inittab` file needed for Case #4b:

```
rb:023456:wait:/usr/robot/bin/setup
```

Case #4c: Using the `build` *Class*

This study modifies a file which exists on the installation machine during package installation. It uses one of three modification methods. The other two methods are shown in Cases #4a and #4b. The file modified is `/sbin/inittab`.

Techniques

This case study shows examples of the following technique:

- Using the `build` class

Approach

This approach to modifying `/sbin/inittab` uses the `build` class. A `build` class file is executed as a shell script and its output becomes the new version of the file being executed. In other words, the data file `inittab` that is delivered with this package will be executed and the output of that execution will become `/sbin/inittab`.

The build class file is executed during package installation and package removal. The argument install is passed to the file if it is being executed at installation time. Notice in the sample build file that installation actions are defined by testing for this argument.

To edit /sbin/inittab using the build class, you must:

- Define the build file in the prototype file.

  The entry for the build file in the prototype file should place it in the build class and define its file type as e. Be certain that the CLASSES parameter in the pkginfo file is defined as build.

- Create the build file.

  The build file shown next performs the following procedures:
  - Edits /sbin/inittab to remove any changes already existing for this package. Notice that the filename /sbin/inittab is hardcoded into the sed command.
  - If the package is being installed, adds the new line to the end of /sbin/inittab. A comment tag is included in this new entry to remind us from where that entry came.
  - Executes init q.

  This solution addresses the drawbacks in case studies #4a and #4b. Only one short file is needed (beyond the pkginfo and prototype files). The file works with multiple instances of a package since the $PKGINST parameter is used, and no postinstall script is required since init q can be executed from the build file.

Sample Files

The following example file shows the pkginfo file for Case #4c:

```
PKG=case4c
NAME=Case Study #4c
CATEGORY=applications
BASEDIR=/
ARCH=SPARC
VERSION=Version 1d05
CLASSES=build
```

This example file shows the `prototype` file for Case #4c:

```
i pkginfo
e build /sbin/inittab ? ? ?
```

The following example file shows the `build` file for Case #4c:

```
# PKGINST parameter provided by installation service
# remove all entries from the existing table that
# are associated with this PKGINST
sed –e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" /sbin/inittab ||
exit 2
if [ "$1" = install ]
then
# add the following entry to the table
echo "rb:023456:wait:/usr/robot/bin/setup #$PKGINST" ||
exit 2
fi
/sbin/init q ||
exit 2
exit 0
```

Case #5: Using `crontab` *in a Class Action Script*

This case study modifies `crontab` files during package installation.

Techniques

This case study shows examples of the following techniques:

- Using classes and class action scripts
- Using the `crontab` command within a class action script

Approach

You could use the `build` class and follow the approach shown for editing `/sbin/inittab` in case study #4c except that you want to edit more than one file. If you used the `build` class approach, you would need to deliver one for each `cron` file edited. Defining a `cron` class provides a more general approach. To edit a `crontab` file with this approach, you must:

- Define the `cron` files that are edited in the `prototype` file.

  Create an entry in the `prototype` file for each `crontab` file that will be edited. Define the class as `cron` and the file type as `e` for each file. Use the actual name of the file to be edited.

- Create the `crontab` files for the package.

  These files contain the information you want added to the existing `crontab` files of the same name.

- Create an installation class action script for the `cron` class.

  The `i.cron` script, shown in the next figure, performs the following procedures:

  - Determines the user ID.

    This is done by setting the variable *user* to the base name of the `cron` class file being processed. That name equates to the user ID. For example, the basename of `/var/spool/cron/crontabs/root` is root (which is also the user ID).

  - Executes `crontab` using the user ID and the `-l` option.

    Using the `-l` options tells `crontab` to send the contents of the `crontab` for the defined user to the standard output.

  - Pipes the output of the `crontab` command to a `sed` script that removes any previous entries added with this installation technique.
  - Puts the edited output into a temporary file.
  - Adds the data file for the root user ID (that was delivered with the package) to the temporary file and adds a tag so you will know where these entries came from.
  - Executes `crontab` with the same user id and give it the temporary file as input.

- Create a removal class action script for the `cron` class.

  The `removal` script is the same as the installation script except there is no procedure to add information to the `crontab` file.

  These procedures are performed for every file in the `cron` class.

Sample Files

The following example file shows the `pkginfo` file for Case #5:

```
PKG=case5
NAME=Case Study #5
CATEGORY=application
BASEDIR=/
ARCH=SPARC
VERSION=Version 1.0
CLASSES=cron
```

This example file shows the `prototype` file for Case #5:

```
i pkginfo
i i.cron
i r.cron
e cron /var/spool/cron/crontabs/root ? ? ?
e cron /var/spool/cron/crontabs/sys ? ? ?
```

This example file shows the installation class action script for Case #5:

```
# PKGINST parameter provided by installation service
while read src dest
do
user=`basename $dest` ||
exit 2
(crontab -l $user |
sed -e "/#$PKGINST$/d" > /tmp/$$crontab) ||
exit 2
sed -e "s/$/#$PKGINST/" $src >> /tmp/$$crontab ||
exit 2
crontab $user < /tmp/$$crontab ||
exit 2
rm -f /tmp/$$crontab
done
exit 0
```

The following example file shows the removal class action script for Case #5:

```
# PKGINST parameter provided by installation service
while read path
do
user=`basename $path` ||
exit 2
(crontab -l $user |
sed -e "/#$PKGINST$/d" > /tmp/$$crontab) ||
exit 2
crontab $user < /tmp/$$crontab ||
exit 2
rm -f /tmp/$$crontab
done
exit
```

The following two example files show the `crontab` files for Case #5:

```
41,1,21 * * * * /usr/lib/uucp/uudemon.hour > /dev/null
45 23 * * * ulimit 5000; /usr/bin/su uucp -c
"/usr/lib/uucp/uudemon.cleanup" >
/dev/null 2>&1
11,31,51 * * * * /usr/lib/uucp/uudemon.poll > /dev/null
```

```
0 * * * 0-6 /usr/lib/sa/sa1
20,40 8-17 * * 1-5 /usr/lib/sa/sa1
5 18 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

In any situation in which the editing of a group of files will increase total file size by more than 10K or so, it is wise to supply a space file so that `pkgadd` can allow for this increase.

Case #6: Installing a Driver

This package installs a driver.

Techniques

This case study shows examples of the following techniques:

- Installing and loading a driver with a `postinstall` script
- Unloading a driver with a `preremove` script

Approach

To install a driver at the time of installation, you must include the object and configuration files for the driver in the `prototype` file.

In this example, the executable module for the driver is named `buffer`. This is the file on which the `add_drv` command operates. The `buffer.conf` file is used by the `kernel` to help configure the driver.

Looking at the prototype file for this example notice the following:

- Since no special treatment is required for these files, you can put them into the standard none class. The CLASSES parameter is set to none in the pkginfo file example.
- The path names for buffer and buffer.conf begin with the variable $KERNDIR. This variable is set in the request script and allows the administrator to decide where the driver files should be installed. The default directory is /kernel/drv.
- There is an entry for the postinstall script (the script that will perform the driver installation).

- Create a request script.

The main function of the request script is to determine where the installer wants the driver objects to be installed, accomplished by questioning the installer and assigning the answer to the $KERNDIR parameter.

The script ends with a routine to make the two parameters CLASSES and KERNDIR available to the installation environment and the postinstall script.

- Create a postinstall script.

The postinstall script actually performs the driver installation. It is executed after the two files buffer and buffer.conf have been installed. The postinstall file shown for this example performs the following actions:

- Uses add_drv to load the driver into the system
- Creates a link for the device using installf
- Finalizes the installation using installf -f

- Create a preremove script.

The preremove script uses rem_drv to unload the driver from the system, and then removes the link /dev/buffer0.

Sample Files

This example shows the `prototype` file for Case #6:

```
i pkginfo
i request
i postinstall
i preremove
f none $KERNDIR/buffer 444 root root
f none $KERNDIR/buffer.conf 444 root root
```

The following example shows the `pkginfo` file for Case #6:

```
PKG=bufdev
NAME=Buffer Device
CATEGORY=system
BASEDIR=
ARCH=INTEL
VERSION=Software Issue #19
CLASSES=none
```

This example shows the `request` script for Case #6:

```
trap 'exit 3' 15
# determine where driver object should be placed; location
# must be an absolute pathname that is an existing directory
KERNDIR=`ckpath -aoy -d /kernel/drv -p \
"Where do you want the driver object installed"` || exit $?

# make parameters available to installation service, and
# so to any other packaging scripts
cat >$1 <<!

CLASSES='$CLASSES'
KERNDIR='$KERNDIR'
!
exit 0
```

The following example shows the `postinstall` script for Case #6:

```
# KERNDIR parameter provided by 'request' script
err_code=1                    # an error is considered fatal
# Load the module into the system
cd $KERNDIR
add_drv -m '* 0666 root sys' buffer || exit $err_code
# Create a /dev entry for the character node
installf $PKGINST /dev/buffer0=/devices/eisa/buffer*:0 s
installf -f $PKGINST
```

This example shows the `preremove` script for Case #6:

```
err_code=1                    # an error is considered fatal
# Unload the driver
rem_drv buffer || exit $err_code
# remove /dev file
removef $PKGINST /dev/buffer0 ; rm /dev/buffer0
removef -f $PKGINST
```

As usual, `removef` and `installf` must be used judiciously in the `preremove` or `preinstall` script since they modify the package database while it is still in transition. Usually these utilities should be restricted to `postremove` and `postinstall` scripts.

Case #7:Using the `sed` *Class and* `postinstall` *and* `preremove` *Scripts*

This study shows how to use the two mandatory control files (`prototype` and `pkginfo`) and `postinstall`, `sed`, and `preremove` scripts to install a driver. There is also a `copyright` file.

The `prototype` file defines all of the contents of the package, that is, it contains an entry for each package object and for each control file (except itself). This file is discussed last (on page 117), immediately before creating the package itself.

B

The `pkginfo` *File*

The `pkginfo` file describes the characteristics of the package. It also contains installation control information. The file consists of a list of *parameter=value* pairs. This is the `pkginfo` file for the driver example:

```
PKG=SUNWsst
NAME=Simple SCSI Target Driver
VERSION=1
CATEGORY=system
ARCH=sparc
VENDOR=Sun Microsystems
BASEDIR=/opt
```

All of the parameters shown in the example are mandatory. The settings of `PKG`, `VERSION` and `ARCH` together define the package *instance*. The installation utility, `pkgadd`(1M), distinguishes instances by appending an instance number to the package name.

The recommended naming convention for packages is the company stock symbol followed by the package name.

See the man page `pkginfo`(4) for full details of the parameters.

The `sed` *Class Script*

`sed` class scripts enable you to modify files that already exist on the system. The script's name indicates the file that the `sed`(1) instructions in the script are executed against. Instructions after the keyword `!install` are executed during installation (after a `preinstall` script but before a `postinstall` script). Instructions after the keyword `!remove` are executed during package removal, in between the `preremove` and the `postremove` scripts.

In the driver example, a `sed` class script is used to add an entry for the driver to the file `/etc/devlink.tab`. This file is used by `devlinks(1M)` to create symbolic links from `/dev` into `/devices`. This is the `sed` script:

```
# sed class script to modify /etc/devlink.tab
!install
/name=sst;/d
$i\
type=ddi_pseudo;name=sst;minor=characterrsst\\A1

!remove
/name=sst;/d
```

The `postinstall` *Installation Script*

This is a Bourne shell script that's run after all files have been installed and all class scripts have been run. In our example, all the script needs to do is run the `add_drv`(1m) utility:

```
# Postinstallation script for SUNWsst
SAVEBASE=$BASEDIR
BASEDIR=””; export BASEDIR
/usr/sbin/add_drv sst
STATUS=$?
BASEDIR=$SAVEBASE; export BASEDIR
if [ $STATUS -eq 0 ]
then
    exit 20
else
    exit 2
fi
```

`add_drv` uses `BASEDIR`, so the script has to unset `BASEDIR` before running the utility, and restore it afterwards.

One of the actions of `add_drv` is to run `devlinks`, which uses the entry placed in `/etc/devlink.tab` by the `sed` class script to create the `/dev` entries for the driver.

The exit code from `postinstall` is significant. `20` tells `pkgadd` to tell the user to reboot the system (necessary after installing a driver), and `2` tells `pkgadd` to tell the user that the installation partially failed.

The `preremove` *Removal Script*

The preremove script is also a Bourne shell script, and it is executed before any package objects are removed from the system. It undoes the actions of the `postinstall` script. In the case of this driver example, it removes the links in `/dev` and runs `rem_drv`(1m) on the driver.

```
# Pre removal script for the sst driver
echo "Removing /dev entries"
/usr/bin/rm -f /dev/rsst*

echo "Deinstalling driver from the kernel"
SAVEBASE=$BASEDIR
BASEDIR=""; export BASEDIR
/usr/sbin/rem_drv sst
BASEDIR=$SAVEBASE; export BASEDIR

exit
```

The script removes the `/dev` entries itself; the `/devices` entries are removed by `rem_drv`.

The `copyright` *File*

This is a simple ASCII file containing the text of a copyright notice. The notice is displayed at the beginning of package installation exactly as it appears in the file.

```
    Copyright (c) 1992 Drivers-R-Us, Inc.
    10 Device Drive, Thebus, IO 80586

All rights reserved. This product and related documentation is
protected by copyright and distributed under licenses restricting
its use, copying, distribution and decompilation. No part of this
product or related documentation may be reproduced in any form by
any means without prior written authorization of Drivers-R-Us and
its licensors, if any.
```

Creating a Package

The main task in creating a package is to create the `prototype` file. This file specifies the locations of the package objects on both the development and the installation workstations. Before creating the `prototype` file, you must determine the layout of the package objects.

Organize the Package Objects

The first step in creating the package is to organize its contents. There are two ways of doing this:

- *Hierarchical*, where the objects on the development machine are in the same directory structure as they will be after installation.
- *Flat*, where the objects on the development machine are in a single directory. In this case, the `prototype` file contains information on the placement of objects on both the development and installation workstation.

Hierarchical Directory Structure

The source file directory structure must be a mirror of the desired structure on the installation machine:

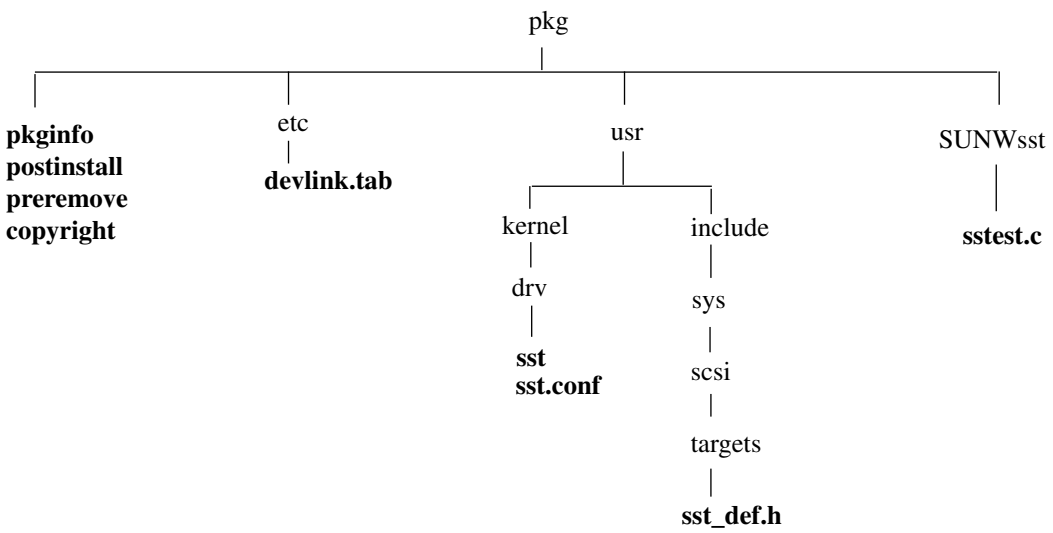

Figure 4-2 Hierarchical Directory Structure

The package objects are installed in the same places as they are in the `pkg` directory above. The driver modules (`sst` and `sst.conf`) are installed into `/usr/kernel/drv` and the include file is installed into `/usr/include/sys/scsi/targets`. `sst`, `sst.conf`, and `sst_def.h` are *fixed* objects. The test program, `sstest.c`, and its directory `SUNWsst` are *relocatable;* their installation location is set by the `BASEDIR` parameter in the `pkginfo` control file (which can be overridden by the administrator during installation).

The remaining components of the package (all the control files) go in the top directory of the package on the development machine, except the `sed` class script. This is called `devlink.tab` after the file it modifies, and goes into `etc`, the directory containing the real `devlink.tab` file.

Flat Directory Structure

It may be more convenient to put all the package objects into a single directory on the development machine. In our example, this is the case, since the installation directory structure is quite sparse.

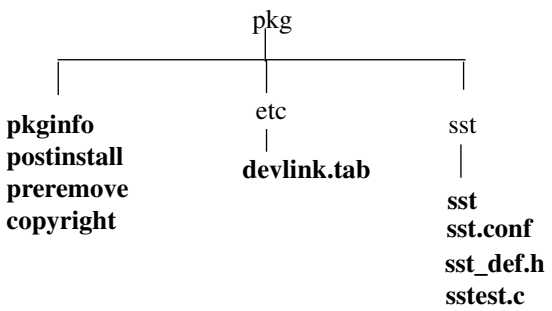

Figure 4-3 Flat Directory Structure

Creating a `prototype` *File for a Hierarchical Directory Structure*

From the `pkg` directory, run the `pkgproto` utility as follows:

```
find usr SUNWsst -print | pkgproto > prototype
```

The entries for the control files in `prototype` have a different format, so you need to insert them manually rather than having `pkgproto` create them for you. The output from the above command looks like this:

```
d none usr 0775 pms mts
d none usr/include 0775 pms mts
d none usr/include/sys 0775 pms mts
d none usr/include/sys/scsi 0775 pms mts
d none usr/include/sys/scsi/targets 0775 pms mts
f none usr/include/sys/scsi/targets/sst_def.h 0444 pms mts
d none usr/kernel 0775 pms mts
d none usr/kernel/drv 0775 pms mts
f none usr/kernel/drv/sst 0664 pms mts
f none usr/kernel/drv/sst.conf 0444 pms mts
d none SUNWsst 0775 pms mts
f none SUNWsst/sstest.c 0664 pms mts
```

This file needs to be modified. Entries are not needed for directories that already exist on the installation machine, the access permissions and ownerships need to be changed, and entries must be added for the control files. Finally, a slash must be prepended to the fixed package objects. This is the final `prototype` file:

```
i pkginfo
i postinstall
i preremove
i copyright
e sed /etc/devlink.tab ? ? ?
f none /usr/include/sys/scsi/targets/sst_def.h 0644 bin bin
f none /usr/kernel/drv/sst 0755 root sys
f none /usr/kernel/drv/sst.conf 0644 root sys
d none SUNWsst 0775 root sys
f none SUNWsst/sstest.c 0664 root sys
```

The questions marks in the entry for the `sed` script indicate that the access permissions and ownership of the existing file on the installation machine should not be changed.

Creating a `prototype` *File for a Flat Directory Structure*

Because the placement of files during installation is not indicated by the development directory structure, file locations must be specified to `pkgproto`. From `sst`, the directory containing the package objects, execute the following command:

```
$ find . -print | pkgproto .=/usr/kernel/drv > ../prototype
```

The parameter to `pkgproto`, `.=/usr/kernel/drv` indicates that the objects in the current directory on the development machine should be installed into the directory `/usr/kernel/drv` on the installation machine. Here's the output from `pkgproto`:

```
d none /usr/kernel/drv 0775 pms mts
f none /usr/kernel/drv/sst=sst 0664 pms mts
f none /usr/kernel/drv/sst.conf=sst.conf 0444 pms mts
f none /usr/kernel/drv/sst_def.h=sst_def.h 0444 pms mts
f none /usr/kernel/drv/sstest.c=sstest.c 0664 pms mts
```

The entries in the `prototype` file specify the locations of the objects on both the development and the installation machines. Note that the source files are relative, so you need to tell `pkgmk` where they are when you run it. This would not be necessary if you had specified `` `pwd`=/usr/kernel/drv `` to `pkgproto` (but the `prototype` entries would be rather long).

The initial `prototype` file must be edited; entries for the control files are added and the access permissions and ownerships need to be changed. Further, the initial `prototype` file shows all files being installed into the same

directory; since this is not what you want, the entries for `sst_def.h` and `sstest.c` must be changed. An entry for the `SUNWsst` directory must also be added. The final file looks like this:

```
i pkginfo
i postinstall
i preremove
i copyright
e sed /etc/devlink.tab ? ? ?
f none /usr/kernel/drv/sst=sst 0755 root sys
f none /usr/kernel/drv/sst.conf=sst.conf 0644 root sys
f none /usr/include/sys/scsi/targets/sst_def.h=sst_def.h 0644
bin bin
d none SUNWsst 0755 root sys
f none SUNWsst/sstest.c=sstest.c 0664 root sys
```

The questions marks in the entry for the `sed` script indicate that the access permissions and ownership of the existing file on the installation machine should not be changed.

Creating the Package

Having organized the package objects, written all the scripts, and created the `prototype` file, you are now ready to actually create the package by running the `pkgmk`(1) utility. This reads the `prototype` file and creates a package that can be installed with `pkgadd`(1m). The syntax for `pkgmk` is slightly different for the flat and hierarchical directory structures.

Creating a Package for a Hierarchical Directory Structure

From the `pkg` directory, run `pkgmk` as follows:

```
$ pkgmk -o -r `pwd` -d spool
```

The `-d` parameter specifies the location of the package to be created. This directory, `spool`, must already exist. The `-r` parameter specifies the root directory for the package objects on the development machine; its value is prepended to the paths in the `prototype` file. The `-o` parameter allows an

existing package to be overwritten. You can safely ignore warnings about missing directory entries for directories that already exist on the installation machine (for example, `/usr/kernel`).

Creating a Package for a Flat Directory Structure

In this case, you must specify the location of the package objects on the development machine with the `-b` parameter:

```
$ pkgmk -o -b `pwd`/sst -d `pwd`/spool -r `pwd`
```

The `-r` option is still needed because the `prototype` entry for the `sed` class script doesn't specify a location on the development machine.

The `pkgmap` *File*

One of the files created by `pkgmk` is the `pkgmap`(4) file. The following example shows the `pkgmap` file for our example package (it is the same for both the hierarchical and the flat directory cases):

```
: 1 120
1 e sed /etc/devlink.tab ? ? ? 218 19258 719022555
1 f none /usr/include/sys/scsi/targets/sst_def.h 0644 bin bin
3623 \
    23380 711071279
1 f none /usr/kernel/drv/sst 0755 root sys 31808 28921 711830351
1 f none /usr/kernel/drv/sst.conf 0644 root sys 326 26818
711830359
1 d none SUNWsst 0775 root sys
1 f none SUNWsst/sstest.c 0664 root sys 3676 19733 711830366
1 i copyright 434 38929 719080369
1 i pkginfo 165 13317 719107352
1 i postinstall 666 55221 719078817
1 i preremove 424 34950 719079244
```

Transferring the Package to Diskette or Tape

The final step in creating a package is to transfer it to a distributable medium, such as diskette or tape. The `pkgtrans(1)` utility performs the transfer. For example, the following command transfers the package `SUNWsst` from the local directory `spool` to a diskette:

```
$ pkgtrans -s `pwd`/spool /vol/dev/rfd0/unlabeled SUNWsst
```

The `-s` option tells `pkgtrans` to convert from file system format to datastream format. `pkgtrans` supports multiple volumes.

The `sed` *Class Script*

`pkgrm` does not run the removal part of the script. You may need to add a line to the `preremove` script to run `sed` directly to remove the entry from `/etc/devlink.tab`.

Note – This example will probably not work correctly if you install it onto a diskless client. In this case, you are better off making the whole package relocatable (install all files into `/opt/SUNWsst`), and then copying the necessary files to the right places in the `postinstall` script. Use `installf`(1M) to put the files into the installation software database. Remember to remove the files in the `preremove` script and also to use `removef`(1M).

Remember to use the appropriate parameters in your path names in order to assure that when installing to a client from a server, the correct files are updated.

`PKG_INSTALL_ROOT`: The root directory of the client.

`BASEDIR`: The location of the files on the client relative to the server.

`CLIENT_BASEDIR`: The location of the files on the client relative to the client.