Chapter 8
Packaging Your Software Applications

An Overview of Software Packaging

This chapter describes how to package software that will be installed on a computer running UNIX System V Release 4 or later releases. The approach to packaging in a UNIX System V Release 4 environment differs from a pre−UNIX System V Release 4 environment. Pre−UNIX System V Release 4 packages deliver information to the system through script actions, but a UNIX System V Release 4 package does this through package information files. A packaging tool, the `pkgmk` command, is provided to help automate package creation. It gathers the components of a package on the development machine, copies them onto the installation medium, and places them into a structure that `pkgadd` recognizes.

This chapter also describes the installation tool, the `pkgadd` command, which copies the package from the installation medium onto a system and performs system housekeeping routines that concern the package. This tool is primarily for the installer but is described here to provide you with a background on the environment into which your packages will be placed and to help you test−install your packages.

The next two sections describe what a package consists of and gives an overview of the structural life cycle of a package (how its structure on your development machine relates to its structure on the installation medium and on the installation machine).

The remaining sections familiarize you with all of the tools, files, and scripts involved in creating a package, provide suggestions for how to approach software packaging, and describe some specific procedures.

A section on set packaging is at the end of this chapter. It describes the new concept of "sets," how packages are related to them, and how some of the packaging commands have been enhanced to support sets.

After reading this chapter, you should study the section entitled Package Installation Case Studies, which provides case studies using the tools and techniques described in this chapter.

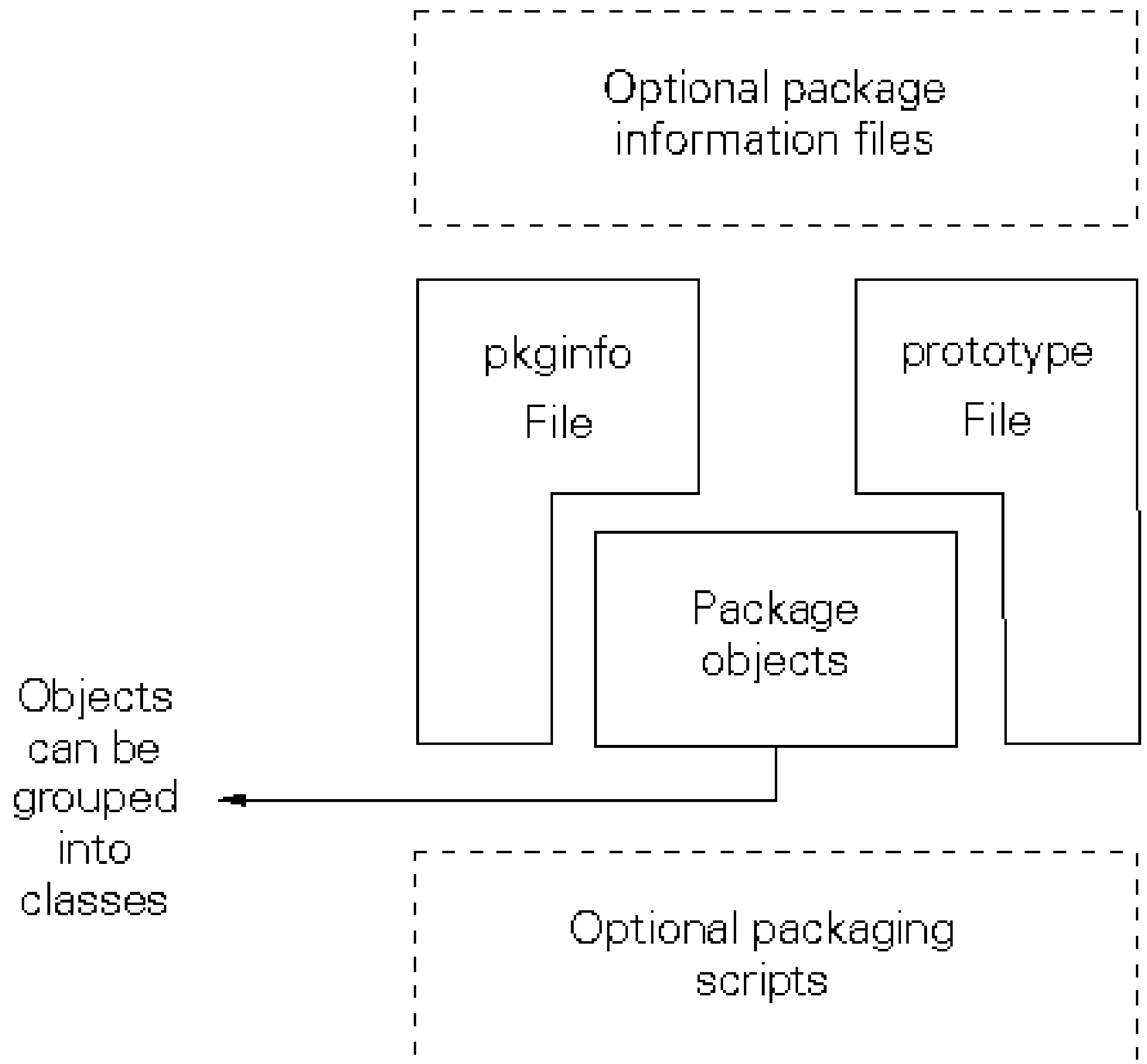
Contents of a Package

A software package is made up of a group of components that together create the software. These components naturally include the executables that comprise the software, but they also include at least two information files and can optionally include other information files and scripts.

As shown in Figure 8−1, a package's contents fall into three categories:

- required components (the `pkginfo` file, the `prototype` file, package objects)
- optional package information files
- optional packaging scripts

Figure 8−1. The Contents of a Package



Required Components

At the very least, a package must contain the following components:

- Package Objects

  These are the objects that make up the software. They can be files (executable or data), directories, or named pipes. Objects can be manipulated in groups during installation by placing them into classes. You will learn more about classes when reading the section Step 3. Placing Objects into Classes.

- The `pkginfo` File

  The `pkginfo` file is a required package information file defining parameter values that describe a package. For example, this file defines values for the package abbreviation, the full package name, and the package architecture.

- The `prototype` File

  The `prototype` file is a required package information file that lists the contents of the package. There is one entry for each deliverable object and this entry consists of several fields of information describing the object. All package components, including the `pkginfo` file, must be listed in the `prototype` file.

Both required package information files are described further in The Package Information Files section and on their respective manual pages.

Optional Package Information Files

There are four optional package information files that you can add to your package:

- The `compver` File

  Defines previous versions of the package that are compatible with this version.

- The `depend` File

  Defines any software dependencies associated with this package.

- The `space` File

  Defines disk space requirements for the target environment beyond that used by objects defined in the `prototype` file (for example, files that will be dynamically created at installation time).

- The `copyright` File

  Defines the text for a copyright message that will be printed on the terminal at the time of package installation or removal.

Every package information file used must have an entry in the `prototype` file. All of these files are described further in the The Package Information Files section and on their respective manual pages.

Optional Installation Scripts

Your package can use three types of installation scripts, although no scripts are required. Many of the tasks executed in a pre–UNIX System V Release 4 installation script are now accomplished automatically by `pkgadd`. However, you may deliver scripts with a UNIX System V Release 4 package to perform customized actions. An installation script must be executable by `sh` (for example, a shell script or executable program). The three script types are the request script (solicits installer input), class action script (defines a set of actions to perform on a group of objects), and the procedure script (defines actions that will occur at particular points during installation).

Packaging scripts are described in detail in ‘‘The Installation Scripts’’ section. Example scripts can be found in the Case Studies.

The Structural Life Cycle of a Package

The material covered in this chapter talks about package object pathnames. While reading, keep in mind that a package object resides in three places while being packaged and installed. To help you avoid confusion, consider which of the three possible locations are being discussed:

- On a development machine Packages originate on a development machine. They can be in the same directory structure on your machine as they will be placed on the installation machine. Or `pkgmk` can locate components on the development machine and give them different pathnames on the installation machine.
- On the installation media When `pkgmk` copies the package components from the development machine to the installation medium, it places them into the structure you have defined in your `prototype` file and a format that `pkgadd` recognizes.
- On the installation machine `pkgadd` copies a package from the installation medium and places it in the structure defined in your `prototype` file. Package objects can be defined as relocatable, meaning the installer can define the actual location of these package objects on the installation machine during installation. Objects with fixed locations are copied to their predefined path.

The Package Creation Tools

The packaging tools are provided to automate package creation and to remove the burden of packaging from the developer. There are three packaging tools:

- From the components of a package on the development machine, `pkgmk` creates a package image in directory structure format.
- `pkgtrans` translates an installable package from one package format to another. The two format types are directory structure and datastream. For example, after having used `pkgmk` to create a package in directory structure format, you might use `pkgtrans` to translate it into datastream format.
- `pkgproto` generates a `prototype` file based on the directory structure of your development area.

Each of these commands is described in the following text and on its manual page.

The pkgmk Command

This command takes all of the package objects residing on the development machine, optionally compresses them, copies them onto the installation medium, and places them into a fixed directory structure. You are not required to know the details of the fixed directory structure since `pkgmk` takes care of the formatting.

Files can be unstructured on the development machine and `pkgmk` will structure them correctly on the medium based on information supplied in the `prototype` file. The installation medium onto which a package is formatted can be what is typically thought of as a medium (a diskette, for example) or it can be a directory on a machine.

`pkgmk` requires the presence of two information files on the development machine, the `prototype` and the `pkginfo` file (other package information files may be present). The `pkginfo` file defines the values for a number of package parameters, such as the package abbreviation and the package name. The `prototype` file provides a complete list of the package contents. `pkgmk` creates the `pkgmap` file, which is the package contents file on the installation medium, by processing the `prototype` file and then adding three fields to each entry.

`pkgmk` follows these steps when processing a package:

1. Processes all of the command lines in the input `prototype` file. (`prototype` command lines can tell `pkgmk`

where to look for package objects, merge other `prototype` files into this one, define default *`mode owner group`* for package objects, and place parameter values in the packaging environment.)

2. Copies the objects of a package onto the installation medium, using the `prototype` file as a listing of contents. If desired, the objects placed on the installation medium may be compressed.
3. Puts the package objects into the proper format.
4. Divides a package into pieces and distributes those pieces on multiple volumes, if necessary.
5. Creates the `pkgmap` file. (the content listing file that is placed on the installation medium). It resembles the `prototype` file except that all command lines are processed, and the `volno`, `size`, `cksum`, and `modtime` fields are added to each entry.

The pkgtrans Command

This command translates a package already created with `pkgmk` from one package format to another. It can make the following translations:

- a fixed directory structure to a datastream
- a datastream to a fixed directory structure
- a fixed directory structure to a fixed directory structure

Note that a package in a fixed directory structure can be in a directory on disk (for example, in a spooling directory) or on a removable device such as a diskette. A datastream can be on any device; for example, on a diskette or a tape.

The pkgproto Command

This command generates a `prototype` file. It scans the paths specified on the command line and creates description line entries for these paths. If the pathname is a directory, an entry for each object in the directory is generated. You can use the `-c` option of the `pkgproto` command to place objects into a particular class.

When you create a `prototype` file with an editor, it does not matter how package components are organized on your development machine. You use the *`path1=path2`* pathname format to define where the files reside on your development machine and where they should be placed on the installation machine. However, when you use `pkgproto` to create your file, your development area must be structured exactly as you want your package to be structured.

The Installation Tools

The installation tools provide capabilities to install and remove packages, create responses to prompts during installation of packages, check the accuracy of installed packages, and display information about software packages. These tools are introduced to you here so that you can understand the environment into which your package will be placed. Manual pages for these tools are provided in the back of this book. The installation tools are:

- `pkgadd` installs a package.
- `pkgrm` removes a package.
- `pkgask` stores answers to an interactive package (one with a request script) in a `response` file. Later, when installing the package, this file may be specified on the `pkgadd` command line so that the package may be installed in noninteractive mode.

- `pkgchk` checks the content and attribute information for an installed package to ensure that it was not corrupted during installation.
- `pkginfo` and `pkgparam` display information about packages.

The system administrator can set parameters that control various aspects of installation in an administration file called the `admin` file. Refer to the manual pages for more information on these commands and on the `admin` file.

The Package Information Files

Each of the six package information files will be described in the following pages. All of these files can be created using any editor. File formats are described in the following text and in full detail in their respective manual pages.

The six package information files are:

- the `pkginfo` file
- the `prototype` file
- the `compver` file
- the `copyright` file
- the `depend` file
- the `space` file

This section also describes the system−generated `pkgmap` file, which `pkgmk` creates and places on the installation medium. It is similar to the `prototype` file.

The pkginfo File

This required package information file defines parameter values that describe characteristics of the package, such as the package abbreviation, full package name, package version, and package architecture. The definitions in this file can set values for all of the installation parameters defined in the `pkginfo` manual page.

Each entry in the file uses the following format to establish the value of a parameter:

```
PARAM="value"
```

Figure 8−2 shows an example `pkginfo` file.

Figure 8−2. Sample pkginfo File

```
PKG="pkgA"
NAME="My Package A"
ARCH="i386"
RELEASE="4.0"
VERSION="2"
VENDOR="MYCOMPANY"
HOTLINE="1-800-677-BUGS"
VSTOCK="0122c3f5566"
CATEGORY="application"
ISTATES="S 2"
RSTATES="S 2"
```

The `pkginfo` and `pkgparam` commands can be used to access information in a `pkginfo` file.

NOTE: Before defining the PKG, ARCH, and VERSION parameters, you need to know how pkgadd defines a package instance and the rules associated with naming a package. Refer to the section Step 2. Defining a Package Instance before assigning values to these parameters.

The prototype File

This required package information file contains a list of the package contents. The pkgmk command uses the prototype file to identify the contents of a package and their location on the development machine when building the package.

You can create this file in two ways. As with all the package information files, you can use an editor to create a file named prototype. It should contain entries following the description given later in this chapter. You can also use the pkgproto command to generate the file automatically. To make use of the second method, you must have a copy of your package on your development machine that is structured exactly as you want it structured on the installation machine and all modes and permissions must be correct. If you are not going to use pkgproto, you do not need a structured copy of your package.

There are two types of entries in the prototype file: description lines and command lines.

The Description Lines

You must create one description line for each deliverable object that consists of several fields describing the object. This entry describes such information as mode, owner, and group for the object. You can also use this entry to accomplish the tasks listed below.

- You can override pkgmk's placement of an object on a multiple−part package. (Refer to the section Step 10. Distributing Packages Over Multiple Volumes for more details.)
- You can place objects into classes. (Refer to the section Step 3. Placing Objects into Classes for details.)
- You can tell pkgmk where to find an object in your development directory structure and map that name to the correct placement on the installation machine. (Refer to the section Mapping Development Pathnames to Installation Pathnames for details.)
- You can define an object as relocatable. (Refer to the sections Defining Collectively Relocatable Objects and Defining Individually Relocatable Objects for details.)
- You can define links. (Refer to the section Step 9. Creating the prototype File for details.)

The generic format of the descriptive line is:

[*part*] *ftype class pathname* [*major minor*] [*mode owner group*]

Definitions for each field are as follows:

part
: Designates the part in which an object should be placed. A package can be divided into a number of parts. A part is a collection of files and is the atomic unit by which a package is processed. A developer can choose the criteria for grouping files into a part (for example, by class). If not defined, pkgmk decides in which part the object will be placed.

ftype
: Designates the file type of an object. Example file types are f (a standard executable or data file), d (a directory), l (a linked file), and i (a package information file). (Refer to the prototype manual page for a complete list of file types.)

class
: Defines the class to which an object belongs. All objects must belong to a class. If the object belongs to no

Defines the class to which an object belongs. All objects must belong to a class. If the object belongs to no special class, this field should be defined as `none`.

`pathname`
: Defines the pathname which an object should have on the installation machine. If you do not begin this name with a slash, the object is considered to be relocatable. You can use the form `path1=path2` to map the location of an object on your development machine to the pathname it should have when installed on a machine.

When a package is stored on an installation medium with an `s5` file–system, each member of the pathname is truncated to 14 characters. When the package is installed on the installation machine with a file–system that supports longer file names, such as `sfs` or `ufs` file–systems, the files are restored with their original pathname length. However, on the `s5` file–system, the pathnames remain truncated.

`major/minor`
: Defines the major and minor numbers for a block or character special device.

`mode/owner/group`
: Defines the mode, owner, and group for an object. The mode, owner, and group must be defined or packaging will fail. If not defined, the defaults defined with the `default` command are assigned.

Figure 8–3 shows an example of this file with only description lines.

Figure 8–3. Sample #1 `prototype` File

```
i pkginfo
i request
d 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
```

The Command Lines

There are four types of commands that can be embedded in the `prototype` file. They are:

`search` *`pathnames`*
: Specifies a list of directories (separated by white space) in which `pkgmk` should search when looking for package objects. *`pathnames`* is prepended to the basename of each object in the `prototype` file until the object is located.

> **NOTE:** The `search` command will not work when invoking `pkgmk` with the `-c` option specified to compress all noninformation package files.

`include` *`filename`*
: Specifies the pathname of another `prototype` file that should be merged into this one during processing. (Note that `search` requests do not span `include` files. Each `prototype` file should have its own `search` command defined, if one is needed.)

`default` *`mode owner group`*
: Defines the default *`mode owner group`* that should be used if this information is not supplied in a `prototype` entry that requires the information. (The defaults do not apply to entries in any `include` files. Each `prototype` file should have its own `default` command defined, if one is needed.)

`param=value`
: Places the indicated parameter in the packaging environment. This allows you to expand a variable pathname so that `pkgmk` can locate the object without changing the actual object pathname. (This assignment will not be available in the installation environment.)

A command line must always begin with an exclamation point (`!`). Commands may have variable substitutions embedded within them.

Figure 8–4 shows an example `prototype` file with both description and command lines.

Figure 8–4. Sample #2 `prototype` File

```
!PROJDIR=/usr/myname
!search /usr/myname/bin /usr/myname/src /usr/myname/hdrs
!include $PROJDIR/src/prototype
i pkginfo
i request
d 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
!default 755 root bin
```

The compver File

This package information file defines previous (or future) versions of the package that are compatible with this version. Each line in the file consists of a string defining a version of the package with which the current version is compatible. Since some packages may require installation of a particular version of another software package, compatibility information is extremely crucial. If a package ‘‘A’’ requires version ‘‘1.0’’ of application ‘‘B’’ as a prerequisite, but the customer installing ‘‘A’’ has a new and improved version of ‘‘1.3’’ of ‘‘B’’, the *`compver`* file for ‘‘B’’ must indicate that the new version is compatible with version ‘‘1.0’’ in order for the customer to install package ‘‘A’’. The string must match the definition of the `VERSION` parameter in the `pkginfo` file of the package considered to be compatible. Figure 8–5 shows an example of this file.

Figure 8–5. Sample `compver` File

```
Version 1.3
Version 1.0
```

The copyright File

This package information file contains the text of a copyright message that will be printed on the terminal at the time of package installation or removal. The display is exactly as shown in the file. Figure 8–6 shows an example of this file.

Figure 8–6. Sample `copyright` File

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

The depend File

This package information file defines software dependencies associated with the package. You can define three types of package dependencies with this file:

- a prerequisite package (meaning this package depends on the existence of another package)
- a reverse dependency (meaning another package depends on the existence of this package)
- an incompatible package (meaning your package is incompatible with this one)

The generic format of a line in this file is:

```
type pkg name
     (arch)version
     (arch)version
```

Definitions for each field are as follows:

`type`
: Defines the dependency type. `P` indicates the named package is a prerequisite for installation. `I` indicates the named package is incompatible. `R` indicates a reverse dependency, that is, the named package requires that this package be on the system. This last type should only be used when a pre−UNIX System V Release 4 package (that cannot deliver a depend file) relies on the newer package.

`pkg`
: Indicates the package abbreviation for the package.

`name`
: Specifies the full package name (used for display purposes only).

`(arch)version`
: Defines a particular instance of a package by defining the architecture and version. If `(arch)version` is not supplied, it means the entry refers to any instance of the package.

Figure 8−7 shows an example of this file.

Figure 8−7. Sample `depend` File

```
P acu Advanced C Utilities
 Issue 4 Version 1
P cc C Programming Language
 Issue 4 Version 1 (386)
R vpkg Another Vendor Package
```

The space File

This package information file defines disk space requirements for the target environment beyond that which is used by objects defined in the `prototype` file−−−for example, files that will be dynamically created at installation time. It should define the maximum amount of additional space that a package will require.

The generic format of a line in this file is:

```
pathname blocks inodes
```

Definitions for each field are as follows:

`pathname`
: Names a directory in which there are objects that will require additional space. The pathname may be the mount point for a file−system. Pathnames that do not begin with a slash (/) indicate relocatable directories.

`blocks`
: Defines the number of 512 byte disk blocks required for installation of the files and directory entries contained in the pathname. (Do not include file−system dependent disk usage.)

`inodes`
Defines the number of inodes required for installation of the files and directory entries contained in `name`.

Numbers of blocks or inodes can be negative to indicate that the package will ultimately (after processing by scripts, and so on) take up less space than the installation tool would calculate.

Figure 8−8 shows an example of this file.

Figure 8−8. Sample `space` File

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

The pkgmap File

The `pkgmk` command creates the `pkgmap` file when it processes the `prototype` file. This new file contains all of the information in the `prototype` file plus three new fields for each entry. These fields are `size` (file size in bytes), `cksum` (checksum of file), and `modtime` (last time of modification). All command lines defined in the `prototype` file are executed as `pkgmk` creates the `pkgmap` file. The `pkgmap` file is placed on the installation medium. The `prototype` file is not. Refer to the `pkgmap` manual page for more details about this file.

The Installation Scripts

The `pkgadd` command automatically performs all of the actions necessary to install a package, using the package information files as input. As a result, you do not have to supply any packaging scripts. However, if you want to customize the installation procedures for your package needs, the following three types of scripts can be used:

request script	Solicits administrator interaction during package installation for the purpose of assigning or redefining environment parameter assignments.
class action scripts	Define an action or set of actions that should be applied to a class of files during installation or removal. You define your own classes or you can use one of three standard classes (`sed`, `awk`, and `build`). See the Step 3. Placing Objects into Classes section for details on how to define a class.
procedure scripts	Specifies a procedure to be invoked before or after the installation or removal of a package. The four procedure scripts are `preinstall`, `postinstall`, `preremove`, and `postremove`.

You decide which type of script to use based on when you want the script to execute. To help you with this assessment, script processing is discussed next, followed by a description of parameters available to packaging scripts, how to get information about a package for your scripts, and script exit codes. After that, each type of script is described in detail.

NOTE: All installation scripts must be executable by `sh` (for example, a shell script or an executable program).

Script Processing

You can customize the actions taken during installation by delivering installation scripts with your package. The decision on which type of script to use to meet a need depends upon when the action is needed during the installation process. As a package is installed, pkgadd performs the following steps:

- Executes the request script.

  This is the only point at which your package can solicit input from the installer.

- Executes the preinstall script.
- Installs the package objects.

  Installation occurs class–by–class and class action scripts are executed accordingly. The list of classes operated upon and the order in which they should be installed is initially defined with the CLASSES parameter in your pkginfo file. However, your request script can change the value of CLASSES.

- Executes the postinstall script.

When a package is being removed, pkgrm performs these steps:

- Executes the preremove script.
- Executes the removal class action scripts.

  Removal also occurs class–by–class. As with the installation class action scripts, if more than one removal script exists, they are processed in the reverse order in which the classes were listed in the CLASSES parameter at the time of installation.

- Executes the postremove script.

The request script is not processed at the time of package removal. However, its output (a list of parameter values) is saved and so is available to removal scripts.

Installation Parameters

The following four groups of parameters are available to all installation scripts. Some of the parameters can be modified by a request script, others cannot be modified at all.

- The four system parameters that are generated by the installation software (see below for a description of these). None of these parameters can be modified by a package.
- The 21 standard installation parameters defined in the pkginfo file. Of these, a package can only modify the CLASSES parameter. (The standard installation parameters are described in detail in the pkginfo manual page.
- You can define your own installation parameters by assigning a value to them in the pkginfo file. Such a parameter must be alphanumeric with an initial capital letter. Any of these parameters can be changed by a request script.
- Your request script can define new parameters by assigning values to them and placing them into the installation environment, as shown in Figure 8–9.

The four installation parameters that are generated by installation software are described below:

PATH	Specifies the search list used by sh to find commands; PATH is set to /sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin upon script invocation.
UPDATE	Indicates that the current installation is intended to update the system. Automatically set to true if the package being installed is overwriting a version of itself.

`PKGINST`	Specifies the instance identifier of the package being installed. If another instance of the package is not already installed, the value will be the package abbreviation. Otherwise, it is the package abbreviation followed by a suffix, such as `pkg.1`. (Multiple variations of the same package can reside simultaneously on the installation medium, as well as on the installation machine. Each variation is known as a package instance and assigned an instance identifier. See Step 2. Defining a Package Instance for more details.)
`PKGSAV`	Specifies the directory where files can be saved for use by removal scripts or where previously saved files may be found.

Getting Package Information for a Script

There are two commands that can be used from your scripts to solicit information about a package.

The `pkginfo` command returns information about software packages, such as the instance identifier and package name.

The `pkgparam` command returns values for all parameters or only for the parameters specified.

The pkginfo(1) and pkgparam(1) manual pages give details for these tools.

Exit Codes for Scripts

Each script must exit with one of the following exit codes:

`0`
: Successful completion of script.

`1`
: Fatal error. Installation process is terminated at this point.

`2`
: Warning or possible error condition. Installation will continue. A warning message will be displayed at the time of completion.

`3`
: Script was interrupted and possibly left unfinished. Installation terminates at this point.

`10`
: System should be rebooted when installation of all selected packages is completed. (This value should be added to one of the single−digit exit codes described above.)

`20`
: The system should be rebooted immediately upon completing installation of the current package. (This value should be added to one of the single−digit exit codes described above.)

See the Case Studies for examples of exit codes in installation scripts.

The Request Script

The request script solicits interaction during installation and is the only place where your package can interact directly with the installer. It can be used, for example, to ask the installer if optional pieces of a package should be installed.

The output of a request script must be a list of parameters and their values. This list can include any of the parameters you created in the pkginfo file (not including the 21 standard parameters) and the CLASSES parameter. The list can also introduce parameters that have not been defined elsewhere.

When your request script assigns values to a parameter, it must then make those values available to the installation environment for use by pkgadd and also by other packaging scripts. The following example shows a request script segment that performs this task for the four parameters CLASSES, NCMPBIN, EMACS, and NCMPMAN.

Figure 8−9. Placing Parameters into the Installation Environment

```
# make parameters available to installation service
# and any other packaging script we might have
cat >$1 <<!
CLASSES='$CLASSES'
NCMPBIN='$NCMPBIN'
EMACS='$EMACS'
NCMPMAN='$NCMPMAN'
!
```

Request Script Naming Conventions

There can only be one request script per package and it must be named request.

Request Script Usage Rules

1. The request script is executed as uid=root and gid=other.
2. The request script should not modify any files, with the exception of the "response" file (described below) which is the output of the request script. It is intended only to interact with users and to create a list of parameter assignments based upon that interaction.
3. pkgadd calls the request script with one argument that names the file to which the output of this script will be written. This file is referred to as the response file.
4. The parameter assignments should be added to the installation environment for use by pkgadd and other packaging scripts (as shown in Figure 8−9).
5. System parameters and standard installation parameters, except for the CLASSES parameter, cannot be modified by a request script. Any of the other parameters available can be changed.
6. The format of the output list should be *PARAMETER*="*value*". For example:

   ```
   CLASSES="none class1"
   ```

7. The list should be written to the file named as the argument to the request script.
8. The user's terminal is defined as standard input to the request script.
9. The request script is not executed during package removal. However, the parameter values assigned in the script are saved and are available during removal.

Soliciting User Input in Request Scripts

A tool is provided in the base package for generating full–screen menus for handling user input. This tool is called `menu`, and should be used when user input is solicited in a request script. Using a form description file [seemenu(4)], `menu` generates a full–screen form that can be used for displaying information, entering a selection from a numbered list, or filling out a more complex, multiple–field form. The `menu` tool may also contain help text so that a user can get more information about the current step in the installation.

The `menu` tool output is a file that contains Bourne/Korn shell statements of the form:

```
VARIABLE="value"
```

After the user completes the menu, this output file can be read in and executed in the request script using the shell '.' command. The values obtained may be used to generate the response file, the output of the request script.

The `menu` tool should be used when soliciting user input from the request script. The `menu` tool can also be used in the postinstall script for a package to inform the user about the status of the installation at completion. (See menu(1) and menu(4) for more information.)

The Class Action Script

The class action script defines a set of actions to be executed during installation or removal of a package. The actions are performed on a group of pathnames based on their class definition. (See the Case Studies for examples of class action scripts.)

Class Action Script Naming Conventions

The name of a class action script is based on which class it should operate and whether those actions should occur during package installation or removal. The two name formats are:

- `i.`*`class`* (operates on pathnames in the indicated class during package installation)
- `r.`*`class`* (operates on pathnames in the indicated class during package removal)

For example, the name of the installation script for a class named `class1` would be `i.class1` and the removal script would be named `r.class1`.

Class Action Script Usage Rules

1. Class action scripts are executed as `uid=root` and `gid=other`.
2. If a package spans more than one volume, the class action script will be executed once for each volume that contains at least one file belonging to the class. Consequently, each script must be ''multiply executable.'' This means that executing a script any number of times with the same input must produce the same results as executing the script only once.

   NOTE: The installation service relies upon this condition being met.

3. The script is executed only if there are files in the given class existing on the current volume.
4. `pkgadd` (and `pkgrm`) creates a list of all objects listed in the `pkgmap` file that belong to the class. As a result, a class action script can only act upon pathnames defined in the `pkgmap` and belonging to a particular class.
5. A class action script should never add, remove, or modify a pathname or system attribute that does not appear in the list generated by `pkgadd` unless by use of the `installf` or `removef` command.

   (See the manual pages for details on these two commands and the Case Studies for examples of them in use.)

6. When the class action script executes for the last time (meaning the input pathname is the last path on the last volume containing a file of this class), it is executed with the keyword argument `ENDOFCLASS`. This flag allows

you to include post–processing actions into your script.

Installation of Classes

The following steps outline the system actions that occur when a class is installed. The actions are repeated once for each volume of a package as that volume is being installed.

1. `pkgadd` creates a pathname list.

   `pkgadd` creates a list of pathnames upon which the action script will operate. Each line of this list consists of source and destination pathnames, separated by white space. The source pathname indicates where the object to be installed resides on the installation volume and the destination pathname indicates the location on the installation machine where the object should be installed. The contents of the list is restricted by the following criteria:

   - The list contains only pathnames belonging to the associated class.
   - Directories, named pipes, character/block devices, and symbolic links are included in the list with the source pathname set to `/dev/null`. They are automatically created by `pkgadd` (if not already in existence) and given proper attributes (mode, owner, group) as defined in the `pkgmap` file.
   - Linked files are not included in the list, that is, files where `ftype` is `l`. (`ftype` defines the file type and is defined in the prototype file.) Links in the given class are created in Step 4.
   - If a pathname already exists on the target machine and its contents are no different from the one being installed, the pathname will not be included in the list.

     To determine this, `pkgadd` compares the `cksum`, `modtime`, and `size` fields in the installation software database with the values for those fields in your `pkgmap` file. If they are the same, it then checks the actual file on the installation machine to be certain it really has those values. If the field values are the same and are correct, the pathname for this object will not be included in the list.

2. If there is no class action script, the files associated with the pathnames are copied to the target machine.

   If no class action script is provided for installation of a particular class, the files in the generated pathname list will simply be copied from the volume to the appropriate target location.

3. If there is a class action script, the script is executed.

   The class action script is invoked with standard input containing the list generated in Step 1. If this is the last volume of the package and there are no more objects in this class, the script is executed with the single argument of `ENDOFCLASS`.

4. `pkgadd` performs a content and attribute audit and creates links.

   After successfully executing Step 2 or 3, an audit of both content and attribute information is performed on the list of pathnames. `pkgadd` creates the links associated with the class automatically. Detected attribute inconsistencies are corrected for all pathnames in the generated list.

Removal of Classes

Objects are removed class–by–class. Classes that exist for a package, but are not listed in the `CLASSES` parameter are removed last (for example, an object installed with the `installf` command). Classes that are listed in the `CLASSES` parameter are removed in reverse order. The following steps outline the system actions that occur when a class is removed:

1. `pkgrm` creates a pathname list.

`pkgrm` creates a list of installed pathnames that belong to the indicated class. Pathnames referenced by another package are excluded from the list unless their ftype is `e` (meaning the file may be edited upon installation or removal).

If a pathname is referenced by another package, it will not be removed from the system. However, if it is of ftype `e`, it may be modified to remove information placed in it by the package being removed. The modification should be performed by the removal class action script.

2. If there is no class action script, the pathnames are removed.

   If your package has no removal class action script for the class, all of the pathnames in the list generated by `pkgrm` will be removed.

   > **NOTE:** You should always assign a class for files with an `ftype` of `e` (editable) and have an associated class action script for that class. Otherwise, they will be removed at this point, even if the pathname is shared with other packages.

3. If there is a class action script, the script is executed.

   `pkgrm` invokes the class action script with standard input containing the list generated in Step 1.

4. `pkgrm` performs an audit.

   Upon successful execution of the class action script, knowledge of the pathnames is removed from the system unless a pathname is referenced by another package.

The Special System Classes

The system provides three special classes. They are:

- The `sed` class (provides a method for using `sed` instructions to edit files upon installation and removal).
- The `awk` class (provides a method for using `awk` instructions to edit files upon installation and removal).
- The `build` class (provides a method to construct a file dynamically during installation).

The sed Class Script

The `sed` installation class provides a method of installing and removing objects that require modification to an existing object on the target machine. (The file must have already been installed by another package.) A `sed` class action script delivers `sed` instructions in the format shown in Figure 8–10. You can give instructions that will be executed during either installation or removal. Two commands indicate when instructions should be executed.`sed` instructions that follow the `!install` command are executed during package installation and those that follow the `!remove` command are executed during package removal. It does not matter in which order the commands are used in the file.

The `sed` class action script executes automatically at installation time if a file belonging to class`sed` exists. The name of the `sed` class file should be the same as the name of the file upon which the instructions will be executed.

Figure 8–10. `sed` Script Format

```
# comment, which may appear on any line in the file
!install
# sed(1) instructions which are to be invoked during
# installation of the object
[address [,address]] function [arguments]
```

```
[address [,address]] function [arguments]
   . . .
!remove
# sed(1) instructions to be invoked during the removal process
[address [,address]] function [arguments]
   . . .
```

address, *function*, and *arguments* are as defined in the sed(1) manual page. See Case Studies 5a and 5b for examples of `sed` class action scripts.

The awk Class Script

The `awk` installation class provides a method of installing and removing objects that require modification to an existing object on the target machine (the object must have been previously installed from another package installation). Modifications are delivered as `awk` instructions in an `awk` class action script.

The `awk` class action script executes automatically at the time of installation if a file belonging to class`awk` exists. Such a file contains instructions for the `awk` class script in the format shown in Figure 8−11. Two commands indicate when instructions should be executed. `awk` instructions that follow the `!install` command are executed during package installation and those that follow the `!remove` command are executed during package removal. It does not matter in which order the commands are used in the file.

The name of the `awk` class file should be the same as the name of the file upon which the instructions will be executed.

Figure 8−11. awk Script Format

```
# comment, which may appear on any line in the file
!install
# awk(1) program to install changes
   . . . (awk program)
!remove
# awk1(1) program to remove changes
   . . . (awk program)
```

The file to be modified is used as input to `awk` and the output of the script ultimately replaces the original object. Parameters may not be passed to `awk` using this syntax.

See Case Study 5a for example `awk` class action scripts.

The build Class Script

The `build` class installs or removes objects by executing instructions that create or modify the object file. These instructions are delivered as a `build` class action script.

The name of the instruction file should be the same as the name of the file upon which the instructions will be executed.

The `build` class action script executes automatically at installation time if a file belonging to class`build` exists.

A `build` script must be executable by `sh`. The script's output becomes the new version of the file as it is built.

See Case Study 5c for an example `build` class action script.

The Procedure Script

The procedure script gives a set of instructions that are performed at particular points in installation or removal.

Four possible procedure scripts are described below. (The Case Studies in the back of the chapter show examples of procedure scripts.)

Naming Conventions for Procedure Scripts

The four procedure scripts must use one of the names listed below, depending on when these instructions are to be executed.

- `preinstall` (executes before class installation begins)
- `postinstall` (executes after all volumes have been installed)
- `preremove` (executes before class removal begins)
- `postremove` (executes after all classes have been removed)

Procedure Script Usage Rules

1. Procedure scripts are executed as `uid=root` and `gid=other`.
2. Each installation procedure script must use the `installf` command to notify `pkgadd` that it will add or modify a pathname. After all additions or modifications are complete, this command should be invoked with the `−f` option to indicate all additions and modifications are complete. (See the `installf` manual page and the Case Studies for details and examples.)
3. Each removal procedure script must use the `removef` command to notify `pkgrm` that it will remove a pathname. After removal is complete, this command should be invoked with the `−f` option to indicate all removals have been completed. (See the `removef` manual page and the Case Studies for details and examples.)

NOTE: The `installf` and `removef` commands must be used for the following reasons. If a procedure script physically removes an object from the system, the system's contents database will still contain an entry for that object until the `removef` command is used to remove the entry. Similarly, if a procedure script places an object on the system, it will not be registered in the contents database until the `installf` command is used to register the object.

Basic Steps of Packaging

What steps you take to create a package depend on how customized your package will be; therefore, it is difficult to give you a step−by−step guide on how to proceed. Your first step should be to plan your packaging. For example, you must decide on which package information files and scripts your package needs.

The following list outlines some of the steps you might use in a packaging scenario. Not all of these steps are required and there exists no mandated order for their execution (although you must have all of your package objects together before executing `pkgmk`). The remainder of this chapter gives procedural information for each step.

NOTE: This list, and the following procedures, are intended only as guidelines. These guidelines can not substitute for reading the rest of this chapter to learn what options are available to your package, and do your own individualized planning.

1. Assign a package abbreviation.

   Every package installed in the UNIX System V Release 4 environment must have a package abbreviation.

2. Define a package instance.

   You must decide on values for the three package parameters that will make each package instance unique. (You need to understand what a package instance is, how it is defined, what the instance identifier is, and how to use that identifier. All of this is covered in the section Step 2. Defining a Package Instance.)

3. Place your objects into classes.

   You must decide on what installation classes you are going to use before you can create the `prototype` file and also before you can write your class action scripts.

4. Set up a package and its objects as relocatable.

   Package objects can be delivered with either fixed locations, meaning that their location is defined by the package and cannot be changed, or with relocatable locations, meaning that they have no absolute location requirements. All of a package or parts of a package can be defined as relocatable. You should decide if package objects will have fixed locations or be relocatable before you write any installation scripts and before you create the `prototype` file.

5. Decide which installation scripts your package needs.

   You must assess the needs of your package beyond the actions provided by `pkgadd` and decide on which type of installation scripts will allow you to deliver your customized actions.

6. Define package dependencies

   You must decide if your package has dependencies on other packages and if any other packages depend on yours.

7. Write a copyright message.

   You must decide if your package requires a copyright message to appear as it is being installed (and removed) and, if so, you must write that message.

8. Create the `pkginfo` file.

   You must create a `pkginfo` file before executing `pkgmk`. It defines basic information concerning the package and can be created with any editor as long as it follows the format described earlier in this chapter and in the `pkginfo` manual page.

9. Create the `prototype` file.

   This file is required and must be created before you execute `pkgmk`. It lists all of the objects that belong to a package and information about each object (such as its file type and to which class it belongs). You can create it using any editor and you must follow the format described earlier in this chapter and in the `prototype` manual page. You can also use the `pkgproto` command to generate a `prototype` file.

10. Distribute packages over multiple volumes.

    `pkgmk` automatically distributes packages over multiple volumes. You must decide if you want to leave those calculations up to `pkgmk` or customize package placement on multiple volumes.

11. Create the package.

    Create the package using the `pkgmk` command, which copies objects from the development machine to the installation medium, puts them into the proper structure, and automatically spans them across multiple volumes, if necessary.

    This is always the last step of packaging, unless you want to create a datastream structure for your package. If so, you must execute `pkgtrans` after creating a package with `pkgmk`.

Step 1. Assigning a Package Abbreviation

Each package installed on UNIX System V Release 4.2 must have a package abbreviation assigned to it. This abbreviation is defined with the `PKG` parameter in the `pkginfo` file.

A valid package abbreviation must meet the criteria defined below:

- It must start with an alphabetic character.
- Additional characters may be alphanumeric and contain the two special characters + and `-`.
- It cannot be longer than nine characters.
- Reserved names are `install`, `new`, and `all`.

Step 2. Defining a Package Instance

The same software package can differ by version or architecture or both. Multiple variations of the same package can reside simultaneously on the same machine. Each variation is known as a package instance. `pkgadd` assigns a package identifier to each package instance at the time of installation. The package identifier is the package abbreviation with a numerical suffix. This identifier distinguishes an instance from any other package, including other instances of the same package.

Identifying a Package Instance

Three parameters defined in the `pkginfo` file combine to identify each instance uniquely. You cannot assign identical values for all three parameters for two instances of the same package installed in the same target environment. These parameters are:

- `PKG` (defines the software package abbreviation and remains constant for every instance of a package)
- `VERSION` (defines the software package version)
- `ARCH` (defines the software package architecture)

For example, you might identify two identical versions of a package that run on different hardware as:

Instance #1	Instance #2
PKG="*abbr*"	PKG="*abbr*"
VERSION="release 1"	VERSION="release 1"
ARCH="MX300I"	ARCH="i386"

Two different versions of a package that run on the same hardware might be identified as:

Instance #1	Instance #2
PKG="*abbr*"	PKG="*abbr*"
VERSION="release 1"	VERSION="release 2"
ARCH="i386"	ARCH="i386"

The instance identifier, assigned by `pkgadd`, maps the three pieces of information that identify an instance to one name consisting of the package abbreviation plus a suffix. The first instance of a package installed on a system does not have a suffix and so its instance identifier will be the package abbreviation. Subsequent instances receive a suffix, beginning with `.2`. An instance is given the lowest integer extension available and so may not correspond to the order in which a package was installed. For example, if `mypkg.2` was deleted after `mypkg.3` was installed, the next instance to be added will be named `mypkg.2`. Because the number of instances of a particular package can vary from machine to machine, the instance identifier can also vary.

> **NOTE:** `pkgmk` also assigns an instance identifier to a package as it places it on the installation medium if one or more instances of a package already exists. That identifier bears no relationship to the identifier assigned to the same package on the installation machine.

Accessing the Instance Identifier in Your Scripts

Because the instance identifier is assigned at the time of installation and will differ from machine to machine, you should use the `PKGINST` system parameter to reference your package in your installation scripts.

Step 3. Placing Objects into Classes

Installation classes allow a series of actions to be performed on a group of package objects at the time of their installation or removal. You place objects into a class in the `prototype` file. All package objects must be given a class, although the class of `none` may be used for objects that require no special action.

The installation parameter `CLASSES`, defined in the `pkginfo` file, is a list of classes to be installed (including the `none` class). Objects defined in the `prototype` file that belong to a class not listed in this parameter will not be installed. The actions to be performed on a class (other than simply copying the components to the installation machine) are defined in a class action script. These scripts are named after the class itself.

For example, to define and install a group of objects belonging to a class named `class1`, follow these steps:

1. Define the objects belonging to `class1` as such in their `prototype` file entry. For example,

   ```
   f class1 /usr/src/myfile
   f class1 /usr/src/myfile2
   ```

2. Ensure that the `CLASSES` parameter in the `pkginfo` file has an entry for `class1`. For example,

   ```
   CLASSES="class1 class2 none"
   ```

   > **NOTE:** Package objects cannot be removed by class.

3. Ensure that a class action script exists for this class. An installation script for a class named `class1` would be named `i.class1` and a removal script would be named `r.class1`.

   If you define a class but do not deliver a class action script, the only action taken for that class will be to copy components from the installation medium to the installation machine.

In addition to the classes that you can define, the system provides three standard classes for your use. The `sed` class provides a method for using `sed` instructions to edit files upon package installation and removal. The `awk` class provides a method for using `awk` instructions to edit files upon package installation and removal. The `build` class provides a method to construct a file dynamically during package installation.

Step 4. Making Package Objects Relocatable

Package objects can be delivered either with fixed locations, meaning that their location on the installation machine is defined by the package and cannot be changed, or as relocatable, meaning that they have no absolute location requirements on the installation machine. The location for relocatable package objects is determined during the installation process.

You can define two types of relocatable objects: collectively relocatable and individually relocatable. All collectively relocatable objects are placed relative to the same directory once the relocatable root directory is established. Individually relocatable objects are not restricted to the same directory location as collectively relocatable objects.

Defining Collectively Relocatable Objects

Follow these steps to define package objects as collectively relocatable:

1. Define a value for the `BASEDIR` parameter.

   Put a definition for the `BASEDIR` parameter in your `pkginfo` file. This parameter names a directory where relocatable objects will be placed by default. If you supply no value for `BASEDIR`, no package objects will be considered as collectively relocatable.

2. Define objects as collectively relocatable in the `prototype` file.

   An object is defined as collectively relocatable by using a relative pathname in its entry in the `prototype` file. A relative pathname does not begin with a slash. For example, `src/myfile` is a relative pathname, while `/src/myfile` is a fixed pathname.

   NOTE: A package can deliver some objects with relocatable locations and others with fixed locations.

All objects defined as collectively relocatable will be put under the same root directory on the installation machine. The root directory value will be one of the following (and in this order):

- if the `admin` file contains `basedir=ask` and `pkgadd` was not invoked in non−interactive mode, then the installer's response to `pkgadd` when asked where relocatable objects should be installed (this overrides the value for `BASEDIR` in the package's `pkginfo` file, if any)
- the value of `BASEDIR` as it is defined in the `admin` file used during the `pkgadd` process (the `BASEDIR` value assigned in the `admin` file overrides the value of the `pkginfo` file)
- the value of `BASEDIR` as it is defined in your `pkginfo` file (this value is used only as a default in case the other two possibilities have not supplied a value)
- if the `admin` file contains `basedir=default` and no `BASEDIR` is set in the package's `pkginfo` file, then `BASEDIR` defaults to `/`

Defining Individually Relocatable Objects

A package object is defined as individually relocatable by using a variable in its pathname definition in the `prototype` file. Your request script must query the installer on where such an object should be placed and assign the response value to the variable. `pkgadd` will expand the pathname based on the output of your request script at the time of installation. Case Study 1 shows an example of the use of variable pathnames and the request script needed to solicit a value for the base directory.

Step 5. Writing Your Installation Scripts

You should read the section The Installation Scripts to learn what types of scripts you can write and how to write them. You can also look at the Case Studies to see how the various scripts can be utilized and to see examples. Remember, you are not required to write any installation scripts for a package.

Reserving Additional Space on the Installation Machine

pkgadd assures that there is enough disk space to install your package, based on the object definitions in the pkgmap file. However, sometimes your package will require additional disk space beyond that needed by the objects defined in the pkgmap file. For example, your package might create a file during installation. pkgadd checks for additional space when you deliver a space file with your package. Refer to the section The space File earlier in this chapter or the space manual page for details on the format of this file.

NOTE: Be certain that your space file has an entry in the prototype file. Its file type should be i (for package information file).

Step 6. Defining Package Dependencies

Package dependencies and incompatibilities can be defined with two of the optional package information files. Delivering a compver file lets you name versions of your package that are compatible with the one being installed. Delivering a depend file lets you define three types of dependencies associated with your package. These dependency types are:

- a prerequisite package (meaning your package depends on the existence of another package)
- a reverse dependency (meaning another package depends on the existence of your package)

  NOTE: This type should only be used when a pre–UNIX System V Release 4 package (that cannot deliver a depend file) relies on the newer package.

- an incompatible package (meaning your package is incompatible with this one)

Refer to the sections The depend File and The compver File earlier in this chapter, or the manual pages depend and compver for details on the formats of these files.

NOTE: Be certain that your depend and compver files have entries in the prototype file. The file type should be i (for package information file).

Step 7. Writing a Copyright Message

To deliver a copyright message, you must create a copyright file named copyright. The message will be displayed exactly as it appears in the file (no formatting) as the package is being installed and as it is being removed. Refer to the section The copyright File earlier in this chapter or the copyright manual page for more detail.

NOTE: Be certain that your copyright file has an entry in the prototype file. Its file type should be i (for package information file).

Step 8. Creating the pkginfo File

The pkginfo file establishes values for parameters that describe the package and is a required package component. The format for an entry in this file is:

```
PARAM="value"
```

PARAM can be any of the 21 standard parameters described in the pkginfo manual page. You can also create your own package parameters simply by assigning a value to them in this file. Your parameter names must begin with a capital letter followed by either upper or lowercase letters.

The following five parameters are required:

- PKG (package abbreviation)
- NAME (full package name)
- ARCH (package architecture)
- VERSION (package version)
- CATEGORY (package category)

The CLASSES parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the none class must be defined in the CLASSES parameter before objects belonging to that class will be installed.

> **NOTE:** You can choose to define the value of CLASSES with a request script and not to deliver a value in the pkginfo file.

Step 9. Creating the prototype File

The prototype file is a list of package contents and is a required package component.

You can create the prototype file by using any editor and following the format described in the section The prototype File and in the prototype manual page. You can also use the pkgproto command to create one automatically.

Creating the File Manually

While creating the prototype file, you must at the very least supply the following three pieces of information about an object:

- The object's type

  All of the possible object types are defined in the prototype manual page. f (for a data file), l (for a linked file), and d (for a directory) are examples of object types.

- The object's class

  All objects must be assigned a class. If no special handling is required, you can assign the class none.

- The object's pathname

  The pathname can define a fixed pathname such as /mypkg/src/filename, a collectively relocatable pathname such as src/filename, and an individually relocatable pathname such as $BIN/filename or /opt/$PKGINST/filename.

Creating Links

To define links you must do the following in the prototype entry for the linked object:

1. Define its ftype as l (a link) or s (a symbolic link).
2. Define its pathname with the format *path1=path2* where *path1* is the destination and *path2* is the source file.

Mapping Development Pathnames to Installation Pathnames

If your development area is in a different structure than you want the package to be in on the installation machine, you can use the `prototype` entry to map one pathname to the other. You use the *`path1=path2`* format for the pathname as is used to define links. However, since the `ftype` is not defined as `l` or `s`, *`path1`* is interpreted as the pathname you want the object to have on the installation machine, and *`path2`* is interpreted as the pathname the object has on your development machine.

For example, your project might require a development structure that includes a project root directory and numerous `src` directories. However, on the installation machine you might want all files to go under a package root directory and for all `src` files to be in one directory. So, a file on your machine might be named `/projdir/srcA/filename`. If you want that file to be named `/pkgroot/src/filename` on the installation machine, your `prototype` entry for this file might look like this:

```
f class1 /pkgroot/src/filename=/projdir/srcA/filename
```

Defining Objects for pkgadd to Create

You can use the `prototype` file to define objects that are not actually delivered on the installation medium. `pkgadd` creates objects with the following `ftypes` if they do not already exist at the time of installation:

- `d` (directories)
- `x` (exclusive directories)
- `l` (linked files)
- `s` (symbolically linked files)
- `p` (named pipes)
- `c` (character special device)
- `b` (block special device)

To request that one of these objects be created on the installation machine, you should add an entry for it in the `prototype` file using the appropriate `ftype`.

For example, if you want a directory created on the installation machine, but do not want to deliver it on the installation medium, an entry for the directory in the `prototype` file is sufficient. An entry such as the one shown below will cause the directory to be created on the installation machine, even if it does not exist on the installation medium.

```
d none /directoryA 644 root other
```

Using the Command Lines

There are four types of commands that you can put into your `prototype` file. They allow you to do the following:

- Nest `prototype` files (the `include` command)
- Define directories for `pkgmk` to look in when attempting to locate objects as it creates the package (the `search` command)

  NOTE: This will not work if `pkgmk` is instructed to compress the package.

- Set a default value for `mode owner group mac fixed inherited` (the `default` command). If all or most of your objects have the same values, using the `default` command will keep you from having to define these values for every entry in the `prototype` file.
- Assign a temporary value for variable pathnames to tell `pkgmk` where to locate these relocatable objects on your

machine (with *param*=*value*)

Creating the File Using pkgproto

The `pkgproto` command scans your directories and generates a `prototype` file. `pkgproto` cannot assign `ftypes` of `v` (volatile files), `e` (editable files), or `x` (exclusive directories). You can edit the `prototype` file and add these `ftypes`, as well as perform any other fine–tuning you require (for example, adding command lines or classes).

`pkgproto` writes its output to the standard output. To create a file, you should redirect the output to a file. The examples shown in this section do not perform redirection in order to show you what the contents of the file would like.

Creating a Basic prototype

The standard format of `pkgproto` is

```
pkgproto path [ . . . ]
```

where *path* is the name of one or more paths to be included in the `prototype` file. If *path* is a directory, then entries are created for the contents of that directory as well (everything below that directory).

With this form of the command, all objects are placed into the `none` class and are assigned the same `mode owner group` as exists on your machine. The following example shows `pkgproto` being executed to create a file for all objects in the directory `/home/pkg`:

```
$ pkgproto /home/pkg
d none /home/pkg 755 bin bin
f none /home/pkg/file1 755 bin bin
f none /home/pkg/file2 755 bin bin
f none /home/pkg/file3 755 bin bin
f none /home/pkg/file4 755 bin bin
f none /home/pkg/file5 755 bin bin
$
```

To create a `prototype` file that contains the output of the example above, you would execute `pkgproto /home/pkg > prototype`

NOTE: If no pathnames are supplied when executing `pkgproto`, standard in (stdin) is assumed to be a list of paths. Refer to the `pkgproto` manual page for details on this usage.

Assigning Objects to a Class

You can use the `−c class` option of `pkgproto` to assign objects to a class other than `none`. When using this option, you can only name one class. To define multiple classes in a `prototype` file created by `pkgproto`, you must edit the file after its creation.

The following example is the same as above except the objects have been assigned to `class1`.

```
$ pkgproto −c class1 /home/pkg
d class1 /home/pkg 755 bin bin
f class1 /home/pkg/file1 755 bin bin
f class1 /home/pkg/file2 755 bin bin
f class1 /home/pkg/file3 755 bin bin
f class1 /home/pkg/file4 755 bin bin
f class1 /home/pkg/file5 755 bin bin
$
```

Renaming Pathnames with pkgproto

You can use a *path1=path2* format on the `pkgproto` command line to give an object a different pathname in the `prototype` file than it has on your machine. You can, for example, use this format to define relocatable objects in a `prototype` file created by `pkgproto`.

The following example is like the others shown in this section, except that the objects are now defined as `bin` (instead of `/usr/bin`) and are thus relocatable.

```
$ pkgproto -c class1 /home/pkg=bin
d class1 bin 755 bin bin
f class1 bin/file1 755 bin bin
f class1 bin/file2 755 bin bin
f class1 bin/file3 755 bin bin
f class1 bin/file4 755 bin bin
f class1 bin/file5 755 bin bin
$
```

pkgproto and Links

`pkgproto` detects linked files and creates entries for them in the `prototype` file. If multiple files are linked together, it considers the first path encountered the source of the link.

If you have symbolic links established on your machine but want to generate an entry for that file with an `ftype` of `f` (file), then use the `-i` option of `pkgproto`. This option creates a file entry for all symbolic links.

Step 10. Distributing Packages Over Multiple Volumes

As a packager, you need not worry about placing package components on multiple volumes. `pkgmk` performs the calculations and actions necessary to organize a multiple volume package.

However, you can use the optional `part` field in the `prototype` file to define in which part you want an object to be placed. A number in this field overrides `pkgmk` and forces the placement of the component into the part given in the field. Note that there is a one−to−one correspondence between parts and volumes for removable media.

Step 11. Creating a Package with pkgmk

`pkgmk` takes all of the objects on your machine (as defined in the `prototype` file), puts them in the fixed directory format and copies everything to the installation medium.

To package your software, execute

```
pkgmk [-d device] [-f filename]
```

You must use the `-d` option to name the device onto which the package should be placed. *device* can be a directory pathname or the identifier for a disk. The default device is the installation spool directory.

`pkgmk` looks for a file named `prototype`. You can use the `-f` option to specify a package contents file named something other than `prototype`. This file must be in the `prototype` format.

For example, executing `pkgmk -d diskette1` creates a package based on a file named `prototype` in your current directory. The package will be formatted and copied to the diskette in the device `diskette1`.

Package File Compression

In Release 4.2, the `pkgmk` command has been enhanced to optionally compress package files. If the `-c` option is specified, pkgmk will compress all non–information files. The following exceptions apply.

If, as a result of compression, the size of the file is not reduced, pkgmk will not compress the file. If the pathname for the file in the package's prototype file is a relative pathname, (for example, ../mypkg/foo), the file will not be compressed.

Creating a Package Instance

pkgmk will create a new instance of a package if one already exists on the device to which it is writing. It will assign the package an instance identifier. Use the −o option of pkgmk to overwrite an existing instance of a package rather than to create a new one.

Helping pkgmk Locate Package Contents

The following list describes situations that might require supplying pkgmk with extra information and an explanation of how to do so:

- Your development area is not structured in the same way that you want your package structured.

  You should use the *path1*=*path2* pathname format in your prototype file.

- You have relocatable objects in your package.

  You can use the *path1*=*path2* pathname format in your prototype file, with *path1* as a relocatable name and *path2* a full pathname to that object on your machine.

  You can use the search command in your prototype file to tell pkgmk where to look for objects. (You can not use the −c option with search, however.)

  You can use the −b *basedir* option to define a pathname that informs pkgmk where to find relocatable object names while creating the package. It does this by prepending basedir to relocatable object names while creating the package. For example, executing

  ```
  pkgmk −d /dev/diskette −b usr2/myhome/reloc
  ```

  would look in the directory /usr2/myhome/reloc for any relocatable object in your package.

- You have variable object names.

  You can use the search command in your prototype file to tell pkgmk where to look for objects. (You can not use the −c option with search, however.)

  You can use the *param*="*value*" command in your prototype file to give pkgmk a value to use for the object name variables as it creates your package.

  You can use the *variable*=*value* option on the pkgmk command line to define a temporary value for variable names.

- The root directory on your machine differs from the root directory described in the prototype file (and that will be used on the installation machine).

  You can use the −r *rootpath* option to tell pkgmk to ignore the destination pathnames in the prototype file. Instead, pkgmk prepends *rootpath* to the source pathnames in order to find objects on your machine.

Step 12. Creating a Package with pkgtrans

pkgtrans performs the following package translations:

- a fixed directory structure to a datastream

- a datastream to a fixed directory structure

To perform one of these translations, execute

```
pkgtrans -s device1 device2 [pkg1[,pkg2[ . . . ]]]
```

where `-s` is the option to translate to datastream, *device1* is the name of the device or directory where the package currently resides, *device2* is the name of the device onto which the translated package will be placed, and [*pkg1*[*pkg2* . . .]] is one or more package names. If no package names are given, a menu of all packages residing in *device1* will be displayed and the user asked for a selection.

Creating a Datastream Package

Creating a datastream package requires two steps:

1. Create a package using `pkgmk`.

   Use the default device (the installation spool directory) or name a directory into which the package should be placed. `pkgmk` creates a package in a fixed directory format. Specify the capacity of the device where the datastream will be placed as an argument to the `-l` option.

2. After the software is formatted in fixed directory format and is residing in a spool directory, execute `pkgtrans`.

   This command translates the fixed directory format to the datastream format and places the datastream on the specified medium.

For example, the two steps shown below will create a datastream package.

1. `pkgmk -d spooldir -l 1400`

   (Formats a package into a fixed directory structure and places it in a directory named `spooldir`. Each part of the package will require no more than 1400 blocks.)

2. `pkgtrans -s spooldir 9track package1`

   (Translates the fixed directory format of `package1` residing in the directory `spooldir` into a datastream format. Places the datastream package on the medium in a device named `9track`.)

   OR

3. `pkgtrans -s spooldir diskette package1`

   Similar to number 2 above, except that it places the datastream package on the medium in a device named `diskette`. `pkgtrans` will prompt for additional volumes if the package requires more than one diskette.

Translating a Package Instance

When an instance of the package being translated to fixed directory format already exists on *device2*, `pkgtrans` will not perform the translation. You can use the `-o` option to tell `pkgtrans` to overwrite any existing instances on the destination device and the `-n` option to tell it to create a new instance if one already exists. Note that the above does not apply when *device2* contains a datastream format.

Set Packaging

Sets provide a method of grouping packages together as one installable entity. Usually this is used to group packages that provide a particular feature or set of features. To enable the set capability in UNIX System V Release 4.2, several that provide a particular feature or set of features. To enable the set capability in UNIX System V Release 4.2, several packaging commands have been enhanced. They are

- pkgadd(1M)
- pkginfo(1)
- pkgrm(1M)

Set Installation

For sets, a special–purpose package referred to as a Set Installation Package (SIP) is used. The SIP is used to control the installation of a set's member packages. The SIP's name and package instance name are always the same as those used to identify the set itself. For instance, the SIP controlling the installation of the Foundation Set (`fnd`) is also named Foundation Set (`fnd`). A SIP is distinguished from other packages by the `CATEGORY` parameter "set" in its `pkginfo` file and by the presence of a special type of package information file named setinfo(4). This file is used to convey information about a set's member packages to the software installation tools.

When `pkgadd` recognizes that a SIP is being processed, it sets up special environment variables and makes them available to the SIP's procedure scripts. This allows for a well–defined interface between the scripts and `pkgadd` that enables the SIP scripts to do most of the work when processing set member package selection and interaction. The SIP's request and preinstall scripts are especially designed to use this environment.

Among other things, the SIP's request script utilizes these environment variables to access the `setinfo` file and access the set member packages' request and default response files (if any). After the request script has finished processing, the SIP's preinstall script is then used to pass back to `pkgadd` a list of set member packages selected for installation as part of the set (see Case 7 in the Package Installation Case Studies section of this guide for examples of these scripts).

The following is a list of the environment variables made available to a SIP's procedure scripts.

- `$SETINFO` This environment variable is used to access the `setinfo` file.
- `$REQDIR` This environment variable provides the directory where the set member packages' request and default response files, if any, reside.
- `$RESPDIR` This environment variable contains the name of the directory where processed response files are to be placed. This response file could be the result of having run a set member package's request script (in the case of custom installation) or simply a copy of the default response file provided with the SIP (in the case of automatic installation).
- `$SETLIST` This environment variable is used to pass back to `pkgadd` the list of packages selected for installation as part of the set.

After it has processed a SIP, `pkgadd` adds the set member packages selected (it gets this from the file represented by `$SETLIST` in the installation environment) to the list of packages to be installed and proceeds to install them.

Set Removal

When the package instance specified to the `pkgrm` command is a SIP, `pkgrm` will remove all of the SIP's set member packages in reverse dependency order (opposite the order in which they were installed). After all of its member packages have been removed, the SIP itself is removed from the system.

Set Information Display

For UNIX System V Release 4.2, the `pkginfo` command has been enhanced to display information about sets. Since the name of the set is the same as its SIP, `pkginfo` must distinguish between when it is being requested to provide information on the SIP, from a request asking for information on the set's member packages (not including the SIP).

If `-c set` is specified, `pkginfo` will display information about the SIP, if one was specified on the command line, or on all SIPs if none was specified. If the category `set` is not specified, `pkginfo` will display information about all packages except those whose category is `set`. If the name of a set is specified on the command line, but `-c set` is not, `pkginfo` will display information on all set member packages belonging to that set except for the SIP itself.

The setsize File

The `setsize` file is a set information file that defines disk space requirements for the target environment. It contains information about all of the packages in the set. This file describes the disk space taken up by installed files as well as extra space needed for dynamically created files, as described in each package's `space` file.

The generic format of a line in this file is:

`pkg:pathname blocks inodes`

Definitions for each field are as follows:

- *`pkg`*

  The short, or abbreviated, name of a package in the set. This name describes which package of the set requires the amount of space described by the rest of the data on this line in the `setsize` file.

- *`pathname`*

  Names a directory in which there are objects that will be installed or that will require additional space. The name may be the mount−point for a file system. Names that do not begin with a slash (/) indicate relocatable directories.

- *`blocks`*

  Defines the number of 512−byte disk blocks required for installation of the files and directory entries contained in the pathname. (Do not include filesystem−dependent disk usage).

- *`inodes`*

  Defines the number of inodes required for the installation of the files and directory entries contained in the pathname.

At installation time, the set installation calls `setsizecvt`, which reduces the `setsize` file for a set to a `space` file containing entries for only the packages that are selected. It is this resulting `space` file against which space checking for the set is performed.

The setsizecvt Command

A new command related to sets, called `setsizecvt`, was created for Release 4.2. This command generates files in the `space` format for sets. Before sets were included as packaging objects, the installation tools used `space` files to specify any additional space the packages required, in addition to that listed in the entries in that package's `pkgmap` file.

The `setsizecvt` command was designed to work as simply as possible; the packaging tools process the sets in much the same way they process packages.

Executing in a set's installation directory, `setsizecvt` collects the `space` file (if it exists) and the `setsize` file from each of the packages included in that set. The `setsize` file is a file whose entries are formatted as follows:

pkg:/path/name #blks #inodes

where *pkg* is the short form of the package name, and the rest is the directory and number of blocks and inodes used in that directory. This `setsize` file is created when the sets are created. Typically, the `setsize` file for a given set would be created from the `pkgmap` files for all of the packages in that set.

`setsizecvt` selects those entries in the `setsize` file for packages (in the current set) that the user wants to install. Those entries are then collected in a new file called `space`.

`pkgadd` uses the `space` file to see if there is enough space on the disk to install the set. The `space` file for a set is treated the same way as it is in a package.

Quick Reference to Packaging Procedures

Before beginning any packaging procedure, you must first have planned your packaging needs based on the information presented in this chapter. ''Basic Steps of Packaging'' gives a comprehensive list of possible packaging steps and considerations. This section only covers the required steps.

1. Create a `prototype` file.
   - Create one manually using any editor. There must be one entry for every package component. The format for a `prototype` file entry is:

     [volno] ftype class pathname [major minor] [mode owner group] [mac fixed inherited]

     volno designates the medium volume number on which the object should be placed. If no *volno* is given, `pkgmk` distributes package components across volumes automatically.

     ftype must be one of these object file types:

     `f (standard executable or data file) e (file to be edited upon installation or removal) v (volatile file, contents will change) d (directory) x (exclusive directory) l (linked file) p (named pipe) c (character special device) b (block special device) i (installation script or package information file) s (symbolic link)`

     class defines the class to which the object belongs. Place an object into the class of `none` if no special handling is required.

     pathname defines the pathname of an object. It can be in one of these formats:

     - fixed pathname: */src/myfile*
     - collectively relocatable pathname: *src/myfile* (no beginning slash)
     - individually relocatable pathname: `$BIN/`*myfile*

     This pathname defines where the component should reside on the installation medium and also tells `pkgmk` where to find it on your machine. If these names differ, use the *path1=path2* format for *pathname*, where *path1* is the name it should have on the installation machine and *path2* is the name it has on your machine.

     major minor defines the major and minor numbers for a block or character special device.

     mode owner group defines the mode, owner and group for the object. If not defined, the value of the `default` command is used. If no default value is defined, `644 root other` is assigned.

     You can use four types of command lines in a `prototype` file:

     `search` *pathnames* (defines a search path for `pkgmk` to use when creating your package)

     `include` *filename* (nests `prototype` files) `default` *mode owner group* (defines a default `mode owner group` for objects defined in this `prototype` file)

param=*value* (defines parameter values for `pkgmk`)

All command lines must begin with an exclamation point (`!`).

- Create one using `pkgproto`.

```
pkgproto [−i] [−c class] [path1[=path2] . . . ] > filename
```

where `−i` tells `pkgproto` to record symbolic links with an `ftype` of `f` (not `s`), `−c` defines the class of all objects as *class*, and *path1* defines the object pathname (or names) to be included in the `prototype` file. If *path1* is a directory, entries for all objects in that directory will be generated.

Use the *path1*=*path2* format to give an object a different pathname in the `prototype` file than it has on your machine. *path1* is the pathname where objects can be located on your machine and *path2* is the pathname that should be substituted for those objects.

`pkgproto` writes its output to the standard output. To create a file, you should redirect the output to a file. That file can be named `prototype` (although it is not required).

2. Create a `pkginfo` file.

   Use any editor. Define one entry per line per parameter in this format:

   PARAM="*value*"

   where *PARAM* is the name of one of the standard installation parameters defined in the `pkginfo` manual page and *value* is the value you assign to it.

   You can also define values for your own installation parameters using the same format. Names for parameters that you create must begin with a capital letter and be followed by only lower−case letters.

   The following five parameters are required in every `pkginfo` file: `PKG`, `NAME`, `ARCH`, `VERSION` and `CATEGORY`. No other restrictions apply concerning which parameters or how many parameters you define.

   The `CLASSES` parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the `none` class must be defined in the `CLASSES` parameter before objects belonging to that class will be installed.

3. Execute `pkgmk`.

```
pkgmk [−d device] [−r rootpath] [−b basedir] [−f filename]
```

   where `−d` specifies that the package should be copied onto *device*, `−r` requests that the root directory *rootpath* be used to locate objects on your machine, `−b` requests that *basedir* be prepended to relocatable paths when searching for them on your machine, and `−f` names a file, *filename*, to be used as your `prototype` file. (Other options are described in the `pkgmk` manual page.)

Refer to the procedures in this chapter for details on other, optional packaging steps (including how to use `pkgtrans` to create a package in datastream structure).

Package Installation Case Studies

This section presents packaging case study in order to show packaging techniques such as installing objects conditionally, determining at run time how many files to create, and how to modify 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 it uses 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

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 which 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 in the `prototype` file, shown in Figure 8−13, that all of the object files belong to one of these three classes.

- Initialize the `CLASSES` parameter in the `pkginfo` file as null.

  Normally when you define a class, you want the `CLASSES` parameter to list all classes that will be installed. Otherwise, no objects in that class will be installed. For this example, the parameter is initially set to null. `CLASSES` will be 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. Figure 8−12 shows the `pkginfo` file associated with this package. Notice that the `CLASSES` parameter is set to null.

- Define object pathnames in the `prototype` file with variables.

  These variables will be set by the request script 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:

  - `$NCMPBIN` (defines location for object executables)
  - `$NCMPMAN` (defines location for manual pages)
  - `$EMACS` (defines location for emacs executables)

  Look at the example `prototype` file (Figure 8−13) to see 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, shown in Figure 8−14, asks two questions:

  - Should this part of the package be installed?

When the answer is yes, then the appropriate class name is added to the `CLASSES` parameter. For example, when the question ''Should the manual pages associated with this package be installed'' is answered yes, the class `man` is added to the `CLASSES` parameter.

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

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

Sample Files

Figure 8−12. Case #1 `pkginfo` File

```
PKG='ncmp'
NAME='NCMP Utilities'
CATEGORY='applications,tools'
ARCH='3b2'
VERSION='Release 1.0, Issue 1.0'
CLASSES=''
```

Figure 8−13. Case #1 `prototype` File

```
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
```

Figure 8−14. Case Study #1 Request Script

```
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 $?
fi
# determine if and where manual pages should be installed
ans=`ckyorn \
        -d y \
        -p "Should manual pages associated with this package be installed"
` || exit $?
if [ "$ans" = y ]
then
        CLASSES="$CLASSES man"
        NCMPMAN=`ckpath -d /usr/ncmp/man -aoy \
                -p "Where should manual pages be installed"
        ` || exit $?
fi
# make parameters available to installation service,
# and so to any other packaging scripts
cat >$1 <<!
CLASSES='$CLASSES'
NCMPBIN='$NCMPBIN'
EMACS='$EMACS'
NCMPMAN='$NCMPMAN'
!
exit 0
```

Case #2

This package installs a driver. A set of device nodes associated with that driver needs to be created, but the installer will decide how many nodes to create. After installation, the system needs to be rebooted so that the driver is properly configured.

Techniques

This case study shows examples of the following techniques:

- installing a driver with a postinstall script
- using an exit code to reboot the system
- allowing the installer to define how many device nodes to create at installation time

Approach

To install a driver at the time of installation, you must:

- Include the object and master files for the driver in the `prototype` file.

  In this example, the object file for the driver is a data file named `qz.o`. This is the file on which the standard UNIX system driver install command, `drvinstall`, operates. The `master.d` file is named `qz` and is used by `drvinstall` to help configure the driver.

  Looking at Figure 8−15 (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 (Figure 8−16).
  - The pathname for `qz.o` begins with the variable `$BOOTDIR`. This variable will be set in the request script and allows the administrator to decide where the object file should be installed. The default directory will be `/boot` .
  - There is an entry for the postinstall script (the script that will perform the driver installation).

- Create a request script.

  The request script, shown in Figure 8−17, has two major functions:

  - to determine how many device nodes to create for this driver

    This is accomplished by questioning the installer and then assigning the answer to the parameter `$NDEVICES`. Notice that the data validation tool `ckrange` is used and that it limits the response to a number between 0 and 32. It sets the default number to 8.

    If the installer chooses not to install any devices, the `CLASSES` parameter is set to null. This means that no classes are defined and therefore no objects will be installed.

  - to determine where the installer wants the driver objects to be installed

    This is accomplished by questioning the installer and assigning the answer to the `$BOOTDIR` parameter.

  The script ends with a routine to make the three parameters `CLASSES`, `NDEVICES`, and `BOOTDIR` available to the installation environment and so to the postinstall script.

- Create a postinstall script.

  The postinstall script, shown in Figure 8−18, actually performs the driver installation. It is executed after the two files `qz` and `qz.o` have been installed. The postinstall shown for this example performs the following actions:

  - checks to see if any devices should be installed (if not, it exits)
  - creates the `/dev/qz` directory using the `installf` command (this directory could also be created by putting an entry for it in the `prototype` file)

    executes the `drvinstall` command using the two files installed with this package (the major number is returned to the script at this time)

- calculates the minor numbers for installed devices
- installs the device using `installf`
- creates a link for the device also using `installf`
- finalizes the installation using `installf -f`

- Reboot the system upon installation.

  This is accomplished by exiting from the postinstall script with an exit code of `10`, meaning that the system should be rebooted upon completing an error−free installation.

Sample Files

Figure 8−15. Case #2 `prototype` File

```
i pkginfo
i request
i postinstall
f none $BOOTDIR/qz.o 444 root root
f none /etc/master.d/qz 444 root root
```

Figure 8−16. Case #2 `pkginfo` File

```
PKG='qzdev'
NAME='qz Devices'
CATEGORY='system'
ARCH='3b2'
VERSION='Software Issue #19'
CLASSES='none'
```

Figure 8−17. Case #2 Request Script

```
trap 'exit 3' 15
# determine if and where general executables should be placed
NDEVICES=`ckrange -l0 -u32 -d 8 \
        -p "How many qz devices do you want configured"
` || exit $?
# if user chose to install no devices, don't install anything
if [ $NDEVICES -eq 0 ]
then
        CLASSES=
else
        # determine where driver object should be placed; location
        # must be an absolute pathname which is an existing directory
        BOOTDIR=`ckpath -aoy -d /boot \
                -p "Where do you want driver object installed"
        ` || exit $?
fi
```

```
# make parameters available to installation service,
# and so to any other packaging scripts
cat >$1 <<!
CLASSES='$CLASSES'
NDEVICES='$NDEVICES'
BOOTDIR='$BOOTDIR'
!
exit 0
```

Figure 8−18. Case #2 Postinstall Script

```
# PKGINST parameter provided by installation service
# NDEVICES parameter provided by 'request' script
# BOOTDIR parameter provided by 'request' script
[ $NDEVICES -eq 0 ] && exit 0
err_code=1  # an error is considered fatal
# need to create the /dev/qz directory
installf $PKGINST /dev/qz d 755 root sys ||
        exit $err_code
# install the driver object and determine major device number
majno=`/usr/sbin/drvinstall -m /etc/master.d/qz -d $BOOTDIR/qz.o -v1.0` ||
        exit $err_code
i=00
while [ $i -lt $NDEVICES ]
do
        for j in 0 1 2 3 4 5 6 7
        do
                # calculate minor number based on loop variables
                minno=`expr $i \ ;8 + $j` || exit $err_code
                # install character device with appropriate major/minor
                # device numbers and correct permissions (installf will
                # do all of work here - you need only provide the info!)
                installf $PKGINST /dev/qz/$i$j c $majno $minno 644 root sys ||
                        exit $err_code
                # create a link from /dev/qz/xx to /dev/qzxx
                installf $PKGINST /dev/qz$i$j=/dev/qz/$i$j ||
                        exit $err_code
        done
        i=`expr $i + 1`
        # add leading zero if necessary
        [ $i -le 9 ] && i="0$i"
done
# finalize installation; the installf command will now
# attempt to create the links that was requested above
installf -f $PKGINST || exit $err_code
exit 10  # requests a reboot from user
```

Case #3

This study creates a database file at the time of 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 will be required to install this package properly
- using the `installf` command

Approach

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

- Create three classes.

  This package requires three classes:

  - 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 (Figure 8–20) that none of the pathnames begin with a slash or a variable. This indicates that they are collectively relocatable.

- Calculate the amount of space the database file will require and create a `space` file to deliver with the package. This file notifies `pkgadd` that this package requires extra space and how much extra space. Figure 8–20 shows the `space` file for this package.

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

  The script, shown in Figure 8–22, 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 will simply be removed from the system.

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

  The script, shown in Figure 8–23, makes a copy of the database file before it is deleted during package removal. 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 pathnames to remove. Pathnames 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 any files to a directory named `/tmp`. When all of the pathnames 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

Figure 8−19. Case #3 `pkginfo` File

```
PKG='krazy'
NAME='KrAzY Applications'
CATEGORY='applications'
ARCH='3b2'
VERSION='Version 1'
CLASSES='none cfgdata admin'
```

Figure 8−20. Case #3 `prototype` File

```
i pkginfo
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 none 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
```

Figure 8−21. Case #3 `space` File

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

Figure 8−22. Case #3 Installation Class Action Script (`i.admin`)

```
# 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
```

Figure 8–23. Case #3 Removal Class Action Script (`r.cfgdata`)

```
# 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 #4

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 Figure 8–25 will be displayed on the screen during package installation (and also during package removal).

- Create a `compver` file.

  The `pkginfo` file shown in Figure 8–24 defines this package version as version 3.0. The `compver` file, shown in Figure 8–26, 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 in Figure 8–27 has 11 packages which must already be on the system at installation time.

Sample Files

Figure 8–24. Case #4 `pkginfo` File

```
PKG='case4'
NAME='Case Study #4'
CATEGORY='application'
ARCH='3b2'
VERSION='Version 3.0'
CLASSES='none'
```

Figure 8–25. Case #4 `copyright` File

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

Figure 8–26. Case #4 `compver` File

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

Figure 8–27. Case #4 `depend` File

```
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 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 #5a

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 5b and 5c. The file modified is `/etc/inittab`.

Techniques

This case study shows examples of the following techniques:

- using the `sed` class
- using a postinstall script

Approach

To modify `/etc/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 `/etc/inittab` and so our `sed` script is named `/etc/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). The `prototype` file for this case study is shown in Figure 8–29.

  > **NOTE:** Since the pathname of the `sed` class action script is exactly the same as the file it is intended to edit, these two may not coexist in the same package.

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

  In the case of the example shown in Figure 8–28, `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 `/etc/inittab` that looks the way you need for it to, since `/etc/inittab` has already been installed and is a dynamic file. Because of this, 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 `/etc/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. See Figure 8–30 for an example `sed` script.

- Create a postinstall script.

You need to inform the system that `/etc/inittab` has been modified by executing `init q`. The only place you can perform that action in this example is in a postinstall script. Looking at the example postinstall script, shown in Figure 8−31, you will see that its only purpose is to execute `init q`.

This approach to editing `/etc/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 hard−coded. It would be nice if this value could be based on the package instance so that you could distinguish between the entries you add for each package.

Sample Files

Figure 8−28. Case #5a `pkginfo` File

```
PKG='case5a'
NAME='Case Study #5a'
CATEGORY='applications'
ARCH='3b2'
VERSION='Version 1d05'
CLASSES='sed'
```

Figure 8−29. Case #5a `prototype` File

```
i pkginfo
i postinstall
e sed /etc/inittab=/home/mypkg/inittab.sed ? ? ?
```

Figure 8−30. Case #5a `sed` Script (`/home/mypkg/inittab.sed`)

```
!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
```

Figure 8−31. Case #5a Postinstall Script

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

Case #5b

This study modifies a file which exists on the installation during package installation. It uses one of three modification methods. The other two methods are shown in Cases 5a and 5c. The file modified is `/etc/inittab`.

Techniques

This case study shows examples of the following techniques:

- creating classes
- using installation and removal class action scripts

Approach

To modify `/etc/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 `inittabl` class in the `CLASSES` parameter in the `pkginfo` file (as shown in Figure 8−32).

- Create an `inittab` file.

  This file contains the information for the entry that you will add to `/etc/inittab`. Notice in the `prototype` file (Figure 8−33) that `inittab` is a member of the `inittab` class and has a file type of `e` for editable. Figure 8−36 shows what `inittab` looks like.

- 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 can't just add our text to the end of the file. The script, shown in Figure 8−34, 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 `/etc/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, shown in Figure 8−35, 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 5a. 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 5c shows a more streamlined approach to editing `/etc/inittab` during installation.

Sample Files

Figure 8–32. Case #5b `pkginfo` File

```
PKG='case5b'
NAME='Case Study #5b'
CATEGORY='applications'
ARCH='3b2'
VERSION='Version 1d05'
CLASSES='inittab'
```

Figure 8–33. Case #5b `prototype` File

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

Figure 8–34. Case #5b Installation Class Action Script (`i.inittab`)

```
# 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/$$itab ||
                exit 2
        sed -e "s/$/#$PKGINST" $src >> /tmp/$$itab ||
                exit 2
        mv /tmp/$$itab $dest ||
                exit 2
done
if [ "$1" = ENDOFCLASS ]
then
        /sbin/init q ||
                exit 2
fi
exit 0
```

Figure 8–35. Case #5b Removal Class Action Script (`r.inittab`)

```
# 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
```

Figure 8–36. Case #5b `inittab` File

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

Case #5c

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 5a and 5b. The file modified is `/etc/inittab`.

Techniques

This case study shows examples of the following technique:

- using the `build` class

Approach

This approach to modifying `/etc/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 for which it is named. In other words, the file `inittab` that is delivered with this package will be executed and the output of that execution will become `/etc/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 in Figure 8–38 that installation actions are defined by testing for this argument.

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

- Define the build file in the `prototype` file.

  The `prototype` file entry for the build class file should be of class `build` and file type `e`. Be certain that the `CLASSES` parameter in the `pkginfo` file includes `build`. Figure 8–37 shows the `pkginfo` file for this example and Figure 8–38 shows the `prototype` file.

- Create the `build` file.

  The `build` file shown in Figure 8–39 performs the following procedures:

  - Edits `/etc/inittab` to remove any changes already existing for this package. Notice that the filename `/etc/inittab` is hard–coded into the `sed` command.
  - If the package is being installed, adds the new line to the end of `/etc/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 5a and 5b. Only one file is needed (beyond the `pkginfo` and `prototype` files), that file is short and simple, it 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

Figure 8–37. Case #5c `pkginfo` File

```
PKG='case5c'
NAME='Case Study #5c'
CATEGORY='applications'
ARCH='3b2'
VERSION='Version 1d05'
CLASSES='build'
```

Figure 8–38. Case #5c `prototype` File

```
i pkginfo
e build /etc/inittab=/home/case5c/inittab.build ? ? ?
```

Figure 8–39. Case #5c `build` Script (`/home/case5c/inittab.build`)

```
# PKGINST parameter provided by installation service
# remove all entries from the existing table that
# are associated with this PKGINST
sed -e "/^[^:]*:[^:]*:[^:]*:[^#]*#$PKGINST$/d" /etc/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 #6

This case study modifies a number of `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 `/etc/inittab` in case study 5c 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 will be edited in the `prototype` file.

  Create an entry in the `prototype` file for each `crontab` file which will be edited. Define their class as `cron` and their file type as `e`. Use the actual name of the file to be edited, as shown in Figure 8–41.

- Create the `crontab` files that will be delivered with the package.

  These files contain the information you want added to the existing `crontab` files of the same name. See Figures 8–44 and 8–45 for examples of what these files look like.

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

  The `i.cron` script (Figure 8–42) performs the following procedures:

  - Calculates 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 standard output the contents of the `crontab` for the defined user.

  - Pipes the output of the `crontab` command to a `sed` script that removes any previous entries that have been added using 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 that you will know from where these entries came.

  - 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, shown in Figure 8–43, is the same as the installation script except that there is no procedure to add information to the `crontab` file.

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

Sample Files

Figure 8–40. Case #3 `pkginfo` File

```
PKG='case6'
NAME='Case Study #6'
CATEGORY='application'
ARCH='3b2'
VERSION='Version 1.0'
CLASSES='cron'
```

Figure 8–41. Case #6 `prototype` File

```
i pkginfo
```

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

Figure 8−42. Case #6 Installation Class Action Script (`i.cron`)

```
# 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
```

Figure 8−43. Case #6 Removal Class Action Script (`r.cron`)

```
# 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 0
```

Figure 8−44. Case #6 Root `crontab` File (Delivered with Package)

```
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
```

Figure 8−45. Case #6 Sys `crontab` File (Delivered with Package)

```
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
```

Case #7a

This case study shows an example of creating a Set Installation Package (SIP) that is used to control the installation of a set of packages.

Techniques

This case study shows examples of the following:

- creating a `setinfo` file (Figure 8–46)
- creating a `request` script that processes set member packages selection and type of installation (custom and default, if applicable) (Figure 8–49)
- including the set member packages' `request` and default `response` files in the `prototype` file as file type `i` files, if any, as part of the SIP so that all interaction with the installer is done only during SIP processing (Figure 8–47)
- using the `preinstall` script to pass the selected packages back to `pkgadd` (Figure 8–48)

Approach

- Create a request script to ask the installer how the set should be installed (Figure 8–49).
- Should default installation be performed on this set?

  When the answer is yes, if any of the set's member packages require interaction and if default responses for that interaction have been provided, install the set using the default responses.

  When the answer is no, for each package in the set, prompt as to whether this package should be installed.

  When the answer is yes, if it is interactive (the package has a request script), should default installation of the package be performed?

  If yes, use the default `response` file.

  If no, execute the package's `request` script to obtain the responses.
- When the request script has completed, the `PKGLIST` variable should contain the list of selected set member packages that will be installed on the system. At this time
  - `pkgadd` runs the SIP's `preinstall` script which places the selected set member packages for installation (`$PKGLIST`) into the `setlist` file referenced via the `$SETLIST` variable.
  - `pkgadd` then reads the `setlist` file and inserts the packages listed there into the list of packages to be installed on the system.
  - As each of these packages is processed, if the package is interactive, `pkgadd` will use the response file created earlier so that no prompting for user input occurs except during SIP processing.

Sample Files

Figure 8–46. Case #7a `setinfo` File

```
# Format for the setinfo file.  Field separator is: <tab>
```

```
# pkg  parts default category package_name
# abbr  y/n
pkgw 1 y system  Package W
pkgx 1 y system  Package X
pkgy 2 n system  Package Y
pkgz 1 y system  Package Y
```

Figure 8−47. Case #7a `prototype` File

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright
i pkgw/request=pkgw.request
i pkgw/response=pkgw.response
i pkgx/request=pkgx.request
i pkgx/response=pkgx.response
```

Figure 8−48. Case #7a `preinstall` Script File

```
for PKG in $PKGLIST
do
 echo "$PKG" >>$SETLIST
done
```

Figure 8−49. Case #7a `request` Script File

```
# If <DELETE> is pressed, make sure we exit 77 so pkgadd knows
# no packages were selected for installation.  In this case,
# pkgadd will also not install the SIP itself.
trap 'EXITCODE=77; exit' 2
trap 'exit $EXITCODE' 0
while read pkginst parts default category package_name
do
 echo $pkginst >>/tmp/order$$
 if [ "$default" = "y" ]
 then
  echo $pkginst >>/tmp/req$$
 else
  echo $pkginst >>/tmp/opt$$
 fi
done <$SETINFO
REQUIRED=`cat /tmp/req$$ 2>/dev/null`
OPTIONAL=`cat /tmp/opt$$ 2>/dev/null`
ORDER=`cat /tmp/order$$ 2>/dev/null`
rm -f /tmp/opt$$ /tmp/req$$ /tmp/order$$

HELPMSG="Enter 'y' to run default set installation or enter
'n' to run custom set installation."
PROMPT="Do you want to run default set installation?"
ANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"`|| exit $?
```

```
if [ "$ANS" = "y" ]
then
 # Default installation
 for PKG in $REQUIRED
 do
  PKGLIST="$PKGLIST $PKG"
  if [ -f $REQDIR/$PKG/response ]
  then
   cp $REQDIR/$PKG/response $RESPDIR/$PKG
  fi
 done
 echo "PKGLIST=$PKGLIST" >> $1
else
 # Custom installation of required packages
 for PKG in $REQUIRED
 do
  PKGLIST="$PKGLIST $PKG"
  if [ -f $REQDIR/$PKG/request ]
  then
   PROMPT="Do you want default installation for $PKG?"
   RANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"` || exit $?
   if [ "$RANS" = "y" ]
   then
    cp $REQDIR/$PKG/request $RESPDIR/$PKG
   else
    sh $REQDIR/$PKG/request $RESPDIR/$PKG
   fi
  fi
 done
 # Select which optional packages in set are to be installed
 for PKG in $OPTIONAL
 do
 HELPMSG="Enter 'y' to install $PKG as part of this set
 installation or 'n' to skip installation."
 PROMPT="Do you want to install $PKG?"
 PANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"` || exit $?
 if [ "$PANS" = "y" -o "$PANS" = "" ]
 then
  PKGLIST="$PKGLIST $PKG"
  if [ -f $REQDIR/$PKG/request ]
  then
     PROMPT="Do you want default installation for $PKG?"
     RANS=`ckyorn -d y -p "$PROMPT" -h "$HELPMSG"` || exit $?
     if [ "$RANS" = "y" ]
     then
      cp $REQDIR/$PKG/request $RESPDIR/$PKG
     else
      sh $REQDIR/$PKG/request $RESPDIR/$PKG
     fi
  fi
 fi
 done
 echo "PKGLIST=$PKGLIST" >> $1
fi
```

```
if [ "$PKGLIST" = "" ]
then
 EXITCODE=77
fi
export SETPKGS
```

Case #7b

This study shows an example of how to split one set into two new sets.

Techniques

This case study shows examples of the following:

- breaking up a `setinfo` file into two
- splitting the set member packages' `request` and default `response` files in the original SIP `prototype` file into two

Approach

From the SIP's `setinfo` file

- create two separate `setinfo` files for the two new sets being created
- create two separate `prototype` files for the two new sets being created

Sample Files

Figure 8−50. Case #7b Original `setinfo` File

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg  parts default category package_name
# abbr  y/n
pkgw 1 y system  Package W
pkgx 1 y system  Package X
pkgy 2 n system  Package Y
pkgz 1 y system  Package Y
```

Figure 8−51. Case #7b SIP One New `setinfo` File

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg  parts default category package_name
# abbr  y/n
pkgw 1 y system  Package W
pkgz 1 y system  Package Y
```

Figure 8−52. Case #7b SIP Two New `setinfo` Files

```
# Format for the setinfo file.  Field separator is: <tab>
# pkg  parts default category package_name
# abbr  y/n
```

```
pkgx 1 y system  Package W
pkgz 1 n system  Package Y
```

Figure 8−53. Case #7b Original `prototype` *File*

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright
i pkgw/request=pkgw.request
i pkgx/response=pkgx.response
i pkgy/request=pkgy.request
i pkgz/response=pkgz.response
```

Figure 8−54. Case #7b SIP One New `prototype` *File*

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright
i pkgw/request=pkgw.request
i pkgz/response=pkgz.response
```

Figure 8−55. Case #7b SIP Two New `prototype` *Files*

```
# set packaging files
i pkginfo
i preinstall
i request
i setinfo
i copyright
i pkgx/request=pkgx.request
i pkgy/response=pkgy.response
```