Application Packaging Developer's Guide

Adobe PostScript™

Please Recycle

Contents

Preface

The *Application Packaging Developer's Guide* provides step-by-step instructions and relevant background information for designing, building, and verifying packages. This guide also includes information on and examples of advanced techniques that you may find helpful during the package creation process.

Who Should Use This Book

This book is intended for application developers whose responsibilities include designing and building packages.

Though much of the book is directed towards novice package developers, it also contains information useful to more experienced package developers.

Note - The Solaris operating environment runs on two types of hardware, or platforms—SPARC and IA. The Solaris operating environment runs on both 64–bit and 32–bit address spaces. The information in this document pertains to both platforms and address spaces unless called out in a special chapter, section, note, bullet, figure, table, example, or code example.

How This Book Is Organized

The following table describes the chapters in this book.

Chapter Name	Chapter Description
Chapter 1	Describes package components, package design criteria, and related commands, files, and scripts.
Chapter 2	Describes the process and required tasks for building a package, and provides step-by-step instructions for each task.
Chapter 3	Describes how to add optional features to a package, and provides step-by-step instructions for each.
Chapter 4	Describes how to verify the integrity of a package and transfer a package to a distribution medium.
Chapter 5	Provides case studies for creating packages.
Chapter 6	Describes various advanced package creation techniques.
Glossary	Contains a list of words and phrases found in this book and their definitions

Related Books

The following documentation, available through retail book sellers, may provide additional background information on building System V packages.

- *System V Application Binary Interface*

- *System V Application Binary Interface - SPARC Processor Supplement*

- *System V Application Binary Interface - Intel386 Processor Supplement*

Ordering Sun Documents

The Sun Software Shop stocks select manuals from Sun Microsystems, Inc. You can purchase individual printed manuals and AnswerBook2™ CDs.

For a list of documents and how to order them, visit the Software Shop at `http://www.sun.com/software/shop/`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic conventions used in this book.

TABLE P–1   Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. Do *not* save changes yet.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the
C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Designing a Package

Before you build a package, you need to know which files you need to create and the commands you need to execute. You also need to consider your application software's requirements, and the needs of your customer—the administrators who will be installing your package. This chapter discusses the files, commands, and criteria you should know and think about, *before* building a package.

This is a list of the overview information in this chapter.

- "Where to Find Packaging Tasks" on page 1
- "What Are Packages?" on page 2
- "Package Components" on page 2
- "Things to Think About Before Building a Package" on page 5
- "Packaging Commands, Files, and Scripts" on page 7

Where to Find Packaging Tasks

Use these references to find step-by-step instructions for building and verifying packages.

- "The Process of Building a Package Task Map" on page 11
- "Creating Information Files and Installation Scripts Task Map" on page 41
- "Verifying and Transferring a Package Task Map" on page 71

1

What Are Packages?

Application software is delivered in units called *packages*. A package is a collection of files and directories required for a software product, and is usually designed and built by the application developer after completing the development of the application code. A software product needs to be built into one or more packages so that it can easily be transferred to a distribution medium, be mass produced, and installed by administrators.

Package Components

The components of a package fall into two categories: *package objects*, the application files to be installed, and *control files*, which control how, where, and if the package is installed.

The control files are also divided into two categories: *information files* and *installation scripts*. Some of these control files are required and some are optional.

To package your applications, you must first create the required components, and any optional components, that make up your package. Then you can build the package using the `pkgmk` command.

To build a package, you must provide the following:

- Package objects (the application software files)
- Two required information files (the `pkginfo` and `prototype` files)
- Optional information files and installation scripts

The figure below describes the contents of a package.
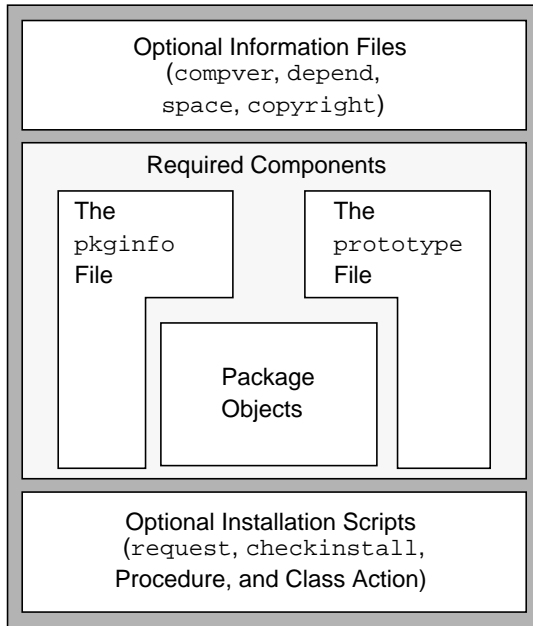
Figure 1–1   The Contents of a Package

Required Package Components

You must create the following components before you build your package:

- Package Objects

  These are the components that make up the application. They can be:

  - Files (executable or data)
  - Directories
  - Named pipes
  - Links
  - Devices

- The `pkginfo` file

  The `pkginfo` file is a required package information file defining parameter values such as the package abbreviation, the full package name, and the package architecture. For more information, see "Creating a `pkginfo` File" on page 14 and the **pkginfo**(4) man page.

> **Note -** There are two **pkginfo**(1) man pages. The first is a section 1 command, which displays information about installed packages. The second is a section 4 file, which describes the characteristics of a package. When accessing the man pages, be sure to specify from which section you want the man page. For example:
> `man -s 4 pkginfo`

- The `prototype` file

  The `prototype` file is a required package information file that lists the components of the package. It describes the location, attributes, and file type for each component within a package.

  In the `prototype` file, there is one entry for each package object, information file, and installation script. An entry consists of several fields of information describing the object. For more information, see "Creating a `prototype` File" on page 20 and the **prototype**(4) man page.

Optional Package Components

Package Information Files

There are four optional package information files you can include in your package:

- The `compver` file

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

- The `depend` file

  Indicates other packages with which this package has special relationships.

- The `space` file

  Defines disk space requirements for the target environment, beyond what is needed by the objects defined in the `prototype` file. For example, additional space might be needed for files that are dynamically created at installation time.

- The `copyright` file

  Defines the text for a copyright message displayed at the time of package installation.

Each package information file should have an entry in the `prototype` file. See "Creating Information Files" on page 43 for more information on creating these files.

Package Installation Scripts

Installation scripts are not required. However, you can provide scripts that perform customized actions during the installation of your package. An installation script has the following characteristics.

- It is composed of Bourne shell commands.
- Its file permissions should be set to 0644.
- It does not need to contain the shell identifier (`#! /bin/sh`).

The four script types are as follows:

- The `request` script

    The `request` script requests input from the administrator installing the package.

- The `checkinstall` script

    The `checkinstall` script performs special file system verification.

Note - The `checkinstall` script is only available with the Solaris™ 2.5 and compatible releases.

- Procedure scripts

    Procedure scripts define actions that occur at particular points during package installation and removal. There are four procedure scripts you can create with these predefined names: `preinstall`, `postinstall`, `preremove`, and `postremove`.

- Class Action scripts

    Class Action scripts define a set of actions to be performed on a group of objects.

See "Creating Installation Scripts" on page 49 for a more information on installation scripts.

Things to Think About Before Building a Package

Before building a package, you need to decide whether your product is going to be made up of one or more packages. (Note that many small packages take longer to install than one big package.) Although it is a good idea to create a single package, it is not always possible. If you decide to build more than one package, you need to determine how to segment the application code. This section provides a list of criteria to use when planning to build packages.

Many of the good packaging criteria present trade-offs among themselves. It will often be difficult to satisfy all requirements equally. These criteria are presented in

order of importance; however, this sequence is meant to serve as a flexible guide depending on the circumstances. Although each of these criteria is important, it is up to you to optimize these requirements to produce a good set of packages.

For more design ideas, see Chapter 6.

Make Packages Installable Remotely

All packages must be *installable remotely*. Being installable remotely means that the administrator installing your package might be trying to install it on a client system, not necessarily to the root file system where the `pkgadd` command is being executed.

Optimize for Client-Server Configurations

Consider the various types of system software configurations (for example, standalone and server) when laying out packages. Good packaging design divides the affected files to optimize installation of each configuration type. For example, the contents of root (`/`) and `usr` should be segmented so that server configurations can easily be supported.

Package by Functional Boundaries

Packages should be self-contained and distinctly identified with a set of functionality. For example, a package containing UFS should contain all UFS utilities and be limited to only UFS binaries.

Packages should be organized from a customer's point of view into functional units.

Package Along Royalty Boundaries

Put code that requires royalty payments due to contractual agreements in a dedicated package or group of packages. Do not disperse the code into more packages than necessary.

Package by System Dependencies

Keep system-dependent binaries in dedicated packages. For example, the kernel code should be in a dedicated package with each implementation architecture corresponding to a distinct package instance. This rule also applies to binaries for

different architectures. For example, binaries for SPARC™ based system would be in one package and binaries for an IA based system would be in another.

Eliminate Overlap in Packages

When constructing packages, eliminate duplicate files whenever possible. Unnecessary duplication of files results in support and version difficulties. If your product has multiple packages, repeatedly compare the contents of these packages for redundancies.

Package Along Localization Boundaries

Localization-specific items should be in their own package. An ideal packaging model would have a product's localizations delivered as one package per locale. Unfortunately, in some cases organizational boundaries may conflict with the functional and product boundaries criteria.

International defaults can also be delivered in a package. This would isolate the files necessary for localization changes and standardize delivery format of localization packages.

Packaging Commands, Files, and Scripts

The purpose of this section is to describe the commands, files, and scripts you might use when manipulating packages. They are described in man pages and in detail throughout this book, in relation to the specific task they perform.

The table below shows the commands available to help you build, verify, install, and obtain information about a package.

TABLE 1–1 Packaging Commands

Function	Command	Description	For More Information, Go To ...
Create packages	`pkgproto`(1)	Generate a `prototype` file for input to the `pkgmk` command	"Creating a `prototype` File With the `pkgproto` Command" on page 26
	`pkgmk`(1)	Create an installable package	"Building a Package" on page 34

TABLE 1–1   Packaging Commands   *(continued)*

Function	Command	Description	For More Information, Go To ...
Install, remove, and transfer packages	**pkgadd**(1M)	Install a software package onto a system	"Installing Software Packages" on page 72
	pkgask(1M)	Store answers to a request script	"Design Rules for request Scripts" on page 54
	pkgtrans(1)	Copy packages onto a distribution medium	"Transferring a Package to a Distribution Medium" on page 84
	pkgrm(1M)	Remove a package from a system	"Removing a Package" on page 83
Obtain information about packages	**pkgchk**(1M)	Check consistency of a software package	"Verifying the Integrity of a Package" on page 75
	pkginfo(1)	Display software package information	"The pkginfo Command" on page 79
	pkgparam(1)	Display package parameter values	"The pkgparam Command" on page 77
Modify installed packages	**installf**(1M)	Incorporate a new package object into an already installed package	"Design Rules for Procedure Scripts" on page 59 and Chapter 5
	removef(1M)	Remove a package object from an already installed package	"Design Rules for Procedure Scripts" on page 59

The table below shows the information files available to help you build a package.

TABLE 1–2 Package Information Files

Files	Description	For More Information, Go To ...
admin(4)	Package installation defaults file	"The Administrative Defaults File" on page 120
compver(4)	Package compatibility file	"Defining Package Dependencies" on page 43
copyright(4)	Package copyright information file	"Writing a Copyright Message " on page 46
depend(4)	Package dependencies file	"Defining Package Dependencies" on page 43
pkginfo(4)	Package characteristics file	"Creating a pkginfo File" on page 14
pkgmap(4)	Package contents description file	"The pkgmap File" on page 35
prototype(4)	Package information file	"Creating a prototype File" on page 20
space(4)	Package disk space requirements file	"Reserving Additional Space on a Target System" on page 47

The table below describes optional installation scripts that you can write that affect if and how a package is installed.

TABLE 1–3   Package Installation Scripts

Scripts	Description	For More Information, Go To ...
request	Solicit information from the installer.	"Writing a request Script" on page 54
checkinstall	Gather file system data.	"Gathering Data With the checkinstall Script" on page 56
preinstall	Perform any custom installation requirements before class installation.	"Writing Procedure Scripts" on page 59
postinstall	Perform any custom installation requirements after all volumes are installed.	"Writing Procedure Scripts" on page 59
preremove	Perform any custom removal requirements before class removal.	"Writing Procedure Scripts" on page 59
postremove	Perform any custom removal requirements after all classes have been removed.	"Writing Procedure Scripts" on page 59
Class Action	Perform a series of actions on a specific group of objects.	"Writing Class Action Scripts" on page 61

Building a Package

This chapter describes a process, as well as the tasks, on how to build a package. Some of these tasks are required and some are optional. The required tasks are the minimum of what you must do to create a package, and are discussed in detail in this chapter. For information on the optional tasks, which enable you to add more features to your package, see Chapter 3 and Chapter 6.

This is a list of the overview information in this chapter.

- "The Process of Building a Package Task Map" on page 11
- "Package Environment Variables" on page 12
- "Creating a `pkginfo` File" on page 14
- "Organizing a Package's Contents" on page 18
- "Creating a `prototype` File" on page 20
- "Building a Package" on page 34

The Process of Building a Package Task Map

The table below describes a process for you to follow when building packages, especially if you are inexperienced at building them. Although it is not mandatory for you to complete the first four tasks in the exact order listed, it will make your package building experience easier if you do. Once you are an experienced package designer, you can shuffle the sequence of these tasks to your preference.

As an experienced package designer, you can automate the package building process using the `make` command and makefiles. For more information, see **make**(1S) and the *Programming Utilities Guide.*

TABLE 2–1 The Process of Building a Package Task Map

Task	Description	For Instructions, Go To ...
1. Create a `pkginfo` File	You must create the `pkginfo` file to describe the characteristics of your package.	"How to Create a `pkginfo` File" on page 17
2. Organize Package Contents	You should arrange your package components into a hierarchical directory structure.	"Organizing a Package's Contents" on page 18
3. Create Information Files	*Optional.* Define package dependencies, include a copyright message, and reserve additional space on a target system.	Chapter 3
4. Create Installation Scripts	*Optional.* Customize the installation and removal processes of a package.	Chapter 3
5. Create a `prototype` File	Describe each object in your package in a `prototype` file.	"Creating a `prototype` File" on page 20
6. Build the Package	Build your package using the `pkgmk` command.	"Building a Package" on page 34
7. Verify and Transfer the Package	Verify the integrity of a package before copying it to a distribution medium.	Chapter 4

Package Environment Variables

You can use variables in the required information files, `pkginfo` and `prototype`, as well as an option to the `pkgmk` command (which is used to build a package). As each of these files and commands are discussed in this chapter, more context-sensitive information on variables is provided. However, before you begin building your package, you should understand the different types of variables and how they can affect a package's successful creation.

There are two types of variables:

- *Build variables*

  Build variables begin with a lowercase letter and are evaluated at *build time* (as the package is being built with the `pkgmk` command).

- *Install variables*

  Install variables begin with an uppercase letter and are evaluated at *install time* (as the package is being installed with the `pkgadd` command).

General Rules on Using Environment Variables

In the `pkginfo` file, a variable definition is of the form *PARAM=value*, where the first letter of *PARAM* is an uppercase letter. These variables are evaluated only at install time, and if any cannot be evaluated, the `pkgadd` command will abort with an error.

In the `prototype` file, a variable definition can take the form !*PARAM=value* or $*variable*. Both *PARAM* and *variable* can begin with either an uppercase or lowercase letter; however, only variables whose values are known at build time will be evaluated. This means that if *PARAM* or *variable* is a build or install variable whose value is not known at build time, the `pkgmk` command will abort with an error.

You can also include the option *PARAM=value* as an option to the `pkgmk` command. This option works the same as in the `prototype` file, except that its scope is global to the entire package. The !*PARAM=value* definition in a `prototype` file is local to that file and the part of the package it defines.

If *PARAM* is an install variable, and *variable* is an install or build variable with a known value, the `pkgmk` command inserts the definition into the `pkginfo` file so that it will be available at install time. However, it will not evaluate *PARAM* in any path names specified in the `prototype` file.

Package Environment Variables Summary

The table below summarizes variable specification formats, location, and scope.

TABLE 2–2 Package Environment Variables Summary

Variable Defined In The ...	Variable Definition Format	Variable Type Being Defined	When The Variable Is Resolved	Where The Variable Is Resolved	The Variable May Be Used As Part Of The ...
`pkginfo` file	*PARAM=value*	Build	Ignored at build time	N/A	None
		Install	Install time	In the `pkgmap` file	*owner, group, path,* or link target
`prototype` file	`!`*PARAM=value*	Build	Build time	In the `prototype` file and any included files	*mode, owner, group,* or *path*
		Install	Build time	In the `prototype` file and any included files	`!search` and `!command` commands only
`pkgmk` command line	*PARAM=value*	Build	Build time	In the `prototype` file	*mode, owner, group,* or *path*
		Install	Build time	In the `prototype` file	`!search` command only
			Install time	In the `pkgmap` file	*owner, group, path,* or link target

Creating a `pkginfo` File

The `pkginfo` file is an ASCII file that describes the characteristics of a package along with information that helps control the flow of installation.

Each entry in the `pkginfo` file is a line that establishes the value of a parameter using the format *PARAM=value*. *PARAM* can be any of the standard parameters

described in **pkginfo**(4), and there is no required order in which the parameters must be specified.

Note - Each *value* can be enclosed with single or double quotation marks (for example, '*value*' or "*value*"). If *value* contains any characters that are considered special to a shell environment, you should use quotation marks. The examples and case studies in this book do not use quotation marks. See **pkginfo**(4) for an example that uses double quotation marks.

You can also create your own package parameters by assigning a value to them in the pkginfo file. Your parameters must begin with a capital letter followed by either uppercase or lowercase letters. An uppercase letter indicates that the parameter (variable) will be evaluated at install time (as opposed to build time). For information on the difference between install and build variables, see "Package Environment Variables" on page 12.

Note - Trailing whitespace after any parameter value is ignored.

There are five parameters that you *must* define in a pkginfo file: PKG, NAME, ARCH, VERSION, and CATEGORY. Three additional parameters: PATH, PKGINST, and INSTDATE are inserted automatically when the package is built. These eight parameters should not be modified. For information on the remaining parameters, see the **pkginfo**(4) man page.

Defining a Package Instance

The same package can have different versions, be compatible with different architectures, or both. Each variation of a package is known as a *package instance*. A package instance is determined by combining the definitions of the PKG, ARCH, and VERSION parameters in the pkginfo file.

The pkgadd command assigns a *package identifier* to each package instance at installation time. The package identifier is the package abbreviation with a numerical suffix, for example SUNWadm.2. This identifier distinguishes a package instance from any other package, including instances of the same package.

Defining a Package Abbreviation (PKG)

A *package abbreviation* is a short name for a package that is defined via the PKG parameter in the pkginfo file, and must:

- Be alphanumeric, but the first cannot be numeric.
- Be nine or fewer characters.
- Not be one of the reserved abbreviations install, new, and all.

> **Note -** The first four characters should be unique to your company, such as your company's stock symbol. For example, packages built by Sun Microsystems™ all have "SUNW" as the first four characters of their package abbreviation.

An example package abbreviation entry in a `pkginfo` file might be:

```
PKG=SUNWcadap
```

Specifying a Package Architecture (`ARCH`)

The `ARCH` parameter in the `pkginfo` file identifies which architectures are associated with the package. The architecture name has a maximum length of 16 alphanumeric characters. If a package is associated with more than one architecture, specify them in a comma-separated list.

For example, a package architecture specification in a `pkginfo` file might be:

```
ARCH=sparc
```

Specifying a Package Instruction Set Architecture (`SUNW_ISA`)

The `SUNW_ISA` parameter in the `pkginfo` file identifies which instruction set architecture is associated with a Sun Microsystems package. The values are:

- `sparcv9`, for a package containing 64–bit objects
- `sparc`, for a package containing 32–bit objects

For example, the `SUNW_ISA` value in a `pkginfo` file for a package containing 64–bit objects would be:

```
SUNW_ISA=sparcv9
```

If `SUNW_ISA` is not set, the default instruction set architecture of the package is set to the value of the `ARCH` parameter.

Specifying a Package Version (`VERSION`)

The `VERSION` parameter in the `pkginfo` file identifies the version of the package. The version has a maximum length of 256 ASCII characters, and cannot begin with a left parenthesis.

An example version specification in a `pkginfo` file might be:

```
VERSION=release 1.0
```

Defining a Package Name (`NAME`)

A *package name* is the full name of the package, which is defined via the `NAME` parameter in the `pkginfo` file.

Because system administrators often use package names to determine whether or not a package needs to be installed, it is important to write clear, concise, and complete package names. Package names should:

- State when a package is needed (for example, to provide certain commands or functionality, or state if it is needed for specific hardware).
- State what the package is used for (for example, the development of device drivers).
- Include a description of the package abbreviation mnemonic, using key words that indicate the abbreviation is a short form of the description (for example, the package name for the package abbreviation `SUNWbnuu` is "Basic Networking UUCP Utilities, (Usr)").
- Name the partition into which the package is installed.
- Use terms consistently with their industry meaning.
- Take advantage of the 256 character limit.

An example package name defined in a `pkginfo` file might be:

```
NAME=Chip designers need CAD application software to design
abc chips.  Runs only on xyz hardware and is installed in the
usr partition.
```

Defining a Package Category (`CATEGORY`)

The `CATEGORY` parameter in the `pkginfo` file specifies in which categories a package belongs. At a minimum, a package must belong to either the `system` or `application` category. Category names:

- Are alphanumeric.
- Have a maximum length of 16 characters.
- Are case insensitive.

If a package belongs to more than one category, specify them in a comma-separated list.

An example `CATEGORY` specification in a `pkginfo` file might be:

```
CATEGORY=system
```

▼ How to Create a `pkginfo` File

1. **Using your favorite text editor, create a file named** `pkginfo`**.**
   You can create this file anywhere on your system.

2. **Edit the file and define the five required parameters.**

The five required parameters are: PKG, ARCH, VERSION, NAME, and CATEGORY. For more information on these parameters, see "Creating a pkginfo File" on page 14.

3. **Add any other parameters that you like to the file.**

   Create your own parameters or see **pkginfo**(4) for information on the standard parameters.

4. **Save your changes and quit the editor.**

Where to Go Next

If you are ready to go to the next task, see "How to Organize A Package's Contents" on page 18.

Example—Creating a pkginfo File

This example shows the contents of a valid pkginfo file, with the five required parameters defined, as well as the BASEDIR parameter. The BASEDIR parameter is discussed in more detail in "The *path* Field" on page 22.

```
PKG=SUNWcadap
NAME=Chip designers need CAD application software to design abc chips.
Runs only on xyz hardware and is installed in the usr partition.
ARCH=sparc
VERSION=release 1.0
CATEGORY=system
BASEDIR=/opt
```

Organizing a Package's Contents

Organize your package objects in a hierarchical directory structure that mimics how you want them to be on the target system after installation. If you do this step before you create a prototype file, you can save yourself some time and effort when creating that file.

▼ How to Organize A Package's Contents

1. **Determine how many packages you need to create and determine which package objects shall be located in each package.**

For help in completing this step, see "Things to Think About Before Building a Package" on page 5.

2. **For each package you need to build, make a directory.**

   You can create this directory anywhere on your system and name it anything you like. The examples in this chapter assume that a package directory has the same name as the package abbreviation.

   ```
   $ cd /home/jane
   $ mkdir SUNWcadap
   ```

3. **For each package, organize package objects into a directory structure beneath their corresponding package directory, which mimics how they will be located on the target system.**

   For example, the CAD application package, SUNWcadap, requires the following directory structure.

   ```
                         /home/jane
                             |
                          SUNWcadap
                             |
        _____
        |          |              |                        |
      demo        lib            man                    srcfiles
        |          |         _____|_____              ___|___
      file1      file2      |            |              |      |
                          windex       man1           file5  file6
                                     ____|____
                                    |        |
                                 file3.1   file4.1
   ```
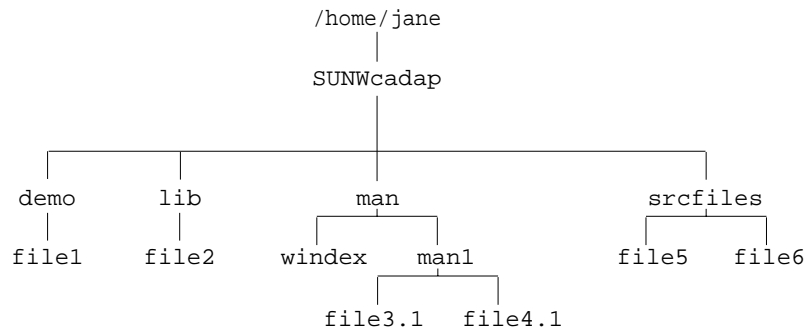
4. **Decide where you will keep your information files and, if appropriate, make a directory to keep them in one location.**

   This example assumes that the example pkginfo file from "How to Create a pkginfo File" on page 17 was created in Jane's home directory.

   ```
   $ cd /home/jane
   $ mkdir InfoFiles
   $ mv pkginfo InfoFiles
   ```

Where to Go Next

If you are ready to go to the next task, see "How to Create a `prototype` File Using the `pkgproto` Command" on page 32.

Creating a `prototype` File

The `prototype` file is an ASCII file used to specify information about the objects in a package. Each entry in the `prototype` file describes a single object, such as a data file, directory, source file, or executable object. Entries in a `prototype` file consist of several fields of information separated by white space. Note that the fields *must* appear in a specific order. Comment lines begin with a pound sign (#) and are ignored.

You can create a `prototype` file with a text editor or by using the `pkgproto` command. When you first create this file, it is probably easier to do so with the `pkgproto` command, because it creates the file based on the directory hierarchy you created previously. If you have not organized your files as described in "Organizing a Package's Contents" on page 18, you have the cumbersome task of creating the `prototype` file from scratch with your favorite text editor. However, even when you create the `prototype` file using the `pkgproto` command, you will most likely need to make modifications to the file with your favorite text editor, so it is important to understand the format and contents of this file.

The Format of the `prototype` File

The format for each line in the `prototype` file is:

part ftype class path major minor group

part	Is an optional, numeric field that enables you to group package objects into parts. The default value is part 1.
ftype	Is a one-character field that specifies the object's type. See "The ftype Field" on page 21.
class	Is the installation class to which the object belongs. See "The *class* Field" on page 22.

path	Is the absolute or relative path name indicating where the package object will reside on the target system. See "The *path* Field" on page 22.
major	Is the major device number for block or character special devices.
minor	Is the minor device number for block or character special devices.
mode	Is the octal mode of the object (for example, 0644). See "The *mode* Field" on page 25.
owner	Is the owner of the object (for example, bin or root). See "The *owner* Field" on page 25.
group	Is the group to which the object belongs (for example, bin or sys). See "The *group* Field" on page 26.

Usually, only the *ftype*, *class*, *path*, *mode*, *owner*, and *group* fields are defined, and are described in the following sections. See **prototype**(4) for additional information on these fields.

The ftype Field

The *ftype*, or file type, field is a one-character field that specifies a package object's type. Valid file types are described in the table below.

TABLE 2–3   Valid File Types in the prototype File

Use *ftype* ...	To Define A ...
f	Standard executable or data file
e	File to be edited upon installation or removal (may be shared by several packages)
v	Volatile file (whose contents are expected to change, like a log file)
d	Directory
x	Exclusive directory accessible only by this package (may contain unregistered logs or database information)
l	Linked file

TABLE 2–3   Valid File Types in the `prototype` File   *(continued)*

Use *ftype* ...	To Define A ...
p	Named pipe
c	Character special device
b	Block special device
i	Information file or installation script
s	Symbolic link

The *class* Field

The *class* field names the class to which an object belongs. Using classes is an optional package design feature, and is discussed in detail in "Writing Class Action Scripts" on page 61.

If you do not use classes, an object belongs to the `none` class, and when you execute the `pkgmk` command to build your package, it will insert the `CLASSES=none` parameter in the `pkginfo` file for you. Files with file type `i` should leave the *class* field blank.

The *path* Field

The *path* field is used to define where the package object will reside on the target system. You may indicate the location with either an absolute path name (for example, `/usr/bin/mail`) or a relative path name (for example, `bin/mail`). Using an absolute path name means that the object's location on the target system is defined by the package and cannot be changed. Package objects with relative path names indicate that the object is *relocatable*.

A *relocatable object* is one that does not need an absolute path location on the target system. Instead, its location is determined during the installation process.

All or some of a package's objects can be defined as relocatable. You should decide if package objects will have fixed locations (such as start-up scripts in `/etc`) or be relocatable before you write any installation scripts and before you create the `prototype` file.

There are two kinds of relocatable objects, *collectively relocatable* and *individually relocatable*.

Collectively Relocatable Objects

Collectively relocatable objects are located relative to a common installation base called the *base directory*. A base directory is defined in the `pkginfo` file, using the `BASEDIR` parameter. For example, a relocatable object in the `prototype` file named `tests/generic` requires that the `pkginfo` file define the default `BASEDIR` parameter. For example:

```
BASEDIR=/opt
```

This means that when the object is installed, it will be located in `/opt/tests/generic`.

Note - `/opt` is the *only* directory to which software that is not part of base Solaris may be delivered.

Use collectively relocatable objects whenever possible. In general, the major part of a package can be relocatable with a few files (such as those in `/etc` or `/var`) specified as absolute. However, if a package contains many different relocations, consider dividing your package into multiple packages, each with a different `BASEDIR` value in its `pkginfo` file.

Individually Relocatable Objects

Individually relocatable objects are not restricted to the same directory location as collectively relocatable objects. To define an individually relocatable object, you need to specify an install variable in the *path* field in the `prototype` file, and then create a `request` script to prompt the installer for the relocatable base directory, or a `checkinstall` script to determine the path name from file system data. For more information on `request` scripts, see "Writing a `request` Script" on page 54 and for information on `checkinstall` scripts, see "How to Gather File System Data" on page 57.

Note - Individually relocatable objects are difficult to manage and should be avoided. This is because they could result in widely scattered package components that may be difficult to isolate when installing multiple versions or architectures of the package. Try to use collectively relocatable objects whenever possible.

Parametric Path Names

A *parametric path name* is a path name that includes a variable specification. For example, `/opt/$PKGINST/`*filename* is a parametric path name because of the `$PKGINST` variable specification. A default value for the variable specification *must* be defined in the `pkginfo` file. The value may then be changed by a `request` or `checkinstall` script.

A variable specification in a path must begin or end the path name, or be bounded by slashes (/). For example, valid parametric path names look like:

```
$PARAM/tests
tests/$PARAM/generic
/tests/$PARAM
```

The variable specification, once defined, may cause the path to be evaluated as absolute or relocatable. For example, given this entry in a `prototype` file:

```
f none $DIRLOC/tests/generic
```

and this entry in the `pkginfo` file:

```
DIRLOC=/myopt
```

the path name, `$DIRLOC/tests/generic`, will evaluate to the *absolute* path name `/myopt/tests/generic`, regardless of whether the `BASEDIR` parameter is set in the `pkginfo` file.

However, if the `pkginfo` file contains these entries

```
DIRLOC=firstcut
BASEDIR=/opt
```

then the path name, `$DIRLOC/tests/generic`, will evaluate to the *relocatable* path name `/opt/firstcut/tests/generic`.

For more information on parametric path names, see "Using Parametric Base Directories" on page 122.

A Brief Word on an Object's Source and Destination Locations

The path name field in the `prototype` file defines where the object will be located on the target system. However, if you did not organize your package's objects on your system in a directory structure that mimics their location on the target system (see "Organizing a Package's Contents" on page 18, then you also need to specify their present location in the `prototype` file.

If your development area is not structured in the same way that you want your package structured, you can use the *path1=path2* format in the *path* field, where *path1* is the location it should have on the target system, and *path2* is the location it has on your system.

You can also use the *path1=path2* path name format with *path1* as a relocatable object name and *path2* a full path name to that object on your system.

> **Note -** *path1* may not contain undefined build variables, but may contain undefined install variables. *path2* may not contain any undefined variables, although both build and install variables may be used. For information on the difference between install and build variables, see "Package Environment Variables" on page 12.

Links must use the *path1= path2* format since they are created by the `pkgadd` command. As a general rule, *path2* of a link should never be absolute, but should instead be relative to the directory portion of *path1*.

An option to using the *path1=path2* format is to use the `!search` command. For more information, see "Providing a Search Path for the `pkgmk` Command" on page 31.

The *mode* Field

The *mode* field may contain an octal number, a question mark (`?`), or a variable specification. An octal number specifies the mode of the object when it is installed on the target system. A `?` means that the mode will be unchanged as the object is installed, implying that the object of the same name already exists on the target system.

A variable specification of the form $mode, where the first letter of the variable must be a lowercase letter, means that this field will be set as the package is built. Note that this variable must be defined at build time in either the `prototype` file or as an option to the `pkgmk` command. For information on the difference between install and build variables, see "Package Environment Variables" on page 12.

Files with file type `i` (information file), `l` (hard link), and `s` (symbolic link), should leave this field blank.

The *owner* Field

The *owner* field may contain a user name, a question mark (`?`), or a variable specification. A user name has a maximum of 14 characters and should be a name that already exists on the target system (such as, `bin` or `root`). A `?` means that the owner will be unchanged as the object is installed, implying that the object of the same name already exists on the target system.

A variable specification can be of the form $*Owner* or $*owner*, where the first letter of the variable is either an uppercase letter or a lowercase letter. If the variable begins with a lowercase letter, it must be defined as the package is built, either in the `prototype` file or as an option to the `pkgmk` command. If the variable begins with an uppercase letter, the variable specification will be inserted into the `pkginfo` file as a default value, and may be redefined at install time via a `request` script. For information on the difference between install and build variables, see "Package Environment Variables" on page 12.

Files with file type `i` (information file) and `l` (hard link) should leave this field blank.

The *group* Field

The *group* field may contain a group name, a question mark (`?`), or a variable specification. A group name has a maximum of 14 characters and should be a name that already exists on the target system (such as, `bin` or `sys`). A `?` means that the group will be unchanged as the object is installed, implying that the object of the same name already exists on the target system.

A variable specification can be of the form $*Group* or $*group*, where the first letter of the variable is either an uppercase letter or a lowercase letter. If the variable begins with a lowercase letter, it must be defined as the package is built, either in the `prototype` file or as an option to the `pkgmk` command. If the variable begins with an uppercase letter, the variable specification will be inserted into the `pkginfo` file as a default value, and may be redefined at install time via a `request` script. For information on the difference between install and build variables, see "Package Environment Variables" on page 12.

Files with file type `i` (information file) and `l` (hard link) should leave this field blank.

Creating a `prototype` File From Scratch

If you want to create a `prototype` file from scratch, you can do so with your favorite text editor, adding one entry per package object. See "The Format of the `prototype` File" on page 20 and **prototype**(4) for more information on the format of the file. However, after you have defined each package object, you might want to include some of the features described in "Adding Functionality to a `prototype` File" on page 29.

Creating a `prototype` File With the `pkgproto` Command

You can use the `pkgproto` command to build a basic `prototype` file, as long as you have organized your package directory structure as described in "Organizing a Package's Contents" on page 18. For example, using the sample directory structure and `pkginfo` file described in previous sections, the commands for creating the `prototype` file are:

```
$ cd /home/jane
$ pkgproto ./SUNWcadap > InfoFiles/prototype
```

The `prototype` file looks like:

```
d none SUNWcadap 0755 jane staff
d none SUNWcadap/demo 0755 jane staff
f none SUNWcadap/demo/file1 0555 jane staff
d none SUNWcadap/srcfiles 0755 jane staff
f none SUNWcadap/srcfiles/file5 0555 jane staff
f none SUNWcadap/srcfiles/file6 0555 jane staff
d none SUNWcadap/lib 0755 jane staff
f none SUNWcadap/lib/file2 0644 jane staff
d none SUNWcadap/man 0755 jane staff
f none SUNWcadap/man/windex 0644 jane staff
d none SUNWcadap/man/man1 0755 jane staff
f none SUNWcadap/man/man1/file4.1 0444 jane staff
f none SUNWcadap/man/man1/file3.1 0444 jane staff
```

Note - The actual owner and group of the person building the package is recorded by the `pkgproto` command. A good technique is to use the `chown -R` and the `chgrp -R` commands, setting the owner and group as intended *before* running the `pkgproto` command.

This `prototype` file is not yet complete. See "Fine-Tuning a `prototype` File Created With the `pkgproto` Command" on page 27 for information on completing this file.

Fine-Tuning a `prototype` File Created With the `pkgproto` Command

Although the `pkgproto` command is useful in creating an initial `prototype` file, it does not create entries for every package object that needs to be defined, or make complete entries. The `pkgproto` command does not:

- Create complete entries for objects with file types `v` (volatile files), `e` (editable files), `x` (exclusive directories), or `i` (information files or installation scripts).
- Support multiple classes with a single invocation.

Therefore, you probably need to modify the `prototype` file to include at least some of these object definitions, possibly redefine path names and other field settings, and include some of the features described in "Adding Functionality to a `prototype` File" on page 29.

Creating Object Entries With File Types `v`, `e`, `x`, and `i`

At the very least, you need to modify the `prototype` file to add objects with file type `i`. If you stored your information files and installation scripts in the first level of your package directory (for example, `/home/jane/SUNWcadap/pkginfo`), then an entry in the `prototype` file would look like:

```
i pkginfo
```

If you did not store your information files and installation scripts in the first level of your package directory, then you need to specify their source location. For example:

```
i pkginfo=/home/jane/InfoFiles/pkginfo
```

Or, you can use the `!search` command to specify the location for the `pkgmk` command to look when building the package. See "Providing a Search Path for the `pkgmk` Command" on page 31 for more information.

To add entries for objects with file types `v`, `e`, and `x`, follow the format described in "The Format of the `prototype` File" on page 20, or refer to **prototype**(4).

Note - Remember to always assign a class to files with a file type of `e` (editable) and have an associated class action script for that class. Otherwise, the files will be removed during package removal, even if the path name is shared with other packages.

Using Multiple Class Definitions

If you use the `pkgproto` command to create your basic `prototype` file, you can assign all of the package objects to the `none` class or one, specific class. As shown in "Creating a `prototype` File With the `pkgproto` Command" on page 26, the basic `pkgproto` command assigns all objects to the `none` class. To assign all objects to a specific class, you can use the −c option. For example:

```
$ pkgproto -c classname /home/jane/SUNWcadap > /home/jane/InfoFiles/prototype
```

If you use multiple classes, you may need to manually edit the `prototype` file and modify the *class* field for each object. If you use classes, you also need to define the `CLASSES` parameter in the `pkginfo` file and write class action scripts. As mentioned previously, using classes is an optional feature and is discussed in detail in "Writing Class Action Scripts" on page 61.

Example—Fine-Tuning a `prototype` File Created Using the `pkgproto` Command

Given the `prototype` file created by the `pkgproto` command in "Creating a `prototype` File With the `pkgproto` Command" on page 26, there are several modifications that need to be made.

- There needs to be an entry for the `pkginfo` file.

- The *path* fields need to be modified to be the *path1=path2* format, since the package source is in `/home/jane`. (Since the package source is a hierarchical directory, and the `!search` command does not search recursively, it may be easier to use the *path1=path2* format.)

- The *owner* and *group* fields should contain the names of existing users and groups on the target system. That is, the owner `jane` will result in an error since this owner is not part of the SunOS™ operating system.

  The modified `prototype` file looks like:

```
i pkginfo=/home/jane/InfoFiles/pkginfo
d none SUNWcadap=/home/jane/SUNWcadap 0755 root sys
d none SUNWcadap/demo=/home/jane/SUNWcadap/demo 0755 root bin
f none SUNWcadap/demo/file1=/home/jane/SUNWcadap/demo/file1 0555 root bin
d none SUNWcadap/srcfiles=/home/jane/SUNWcadap/srcfiles 0755 root bin
f none SUNWcadap/srcfiles/file5=/home/jane/SUNWcadap/srcfiles/file5 0555 root bin
f none SUNWcadap/srcfiles/file6=/home/jane/SUNWcadap/srcfiles/file6 0555 root bin
d none SUNWcadap/lib=/home/jane/SUNWcadap/lib 0755 root bin
f none SUNWcadap/lib/file2=/home/jane/SUNWcadap/lib/file2 0644 root bin
d none SUNWcadap/man=/home/jane/SUNWcadap/man 0755 bin bin
f none SUNWcadap/man/windex=/home/jane/SUNWcadap/man/windex 0644 root other
d none SUNWcadap/man/man1=/home/jane/SUNWcadap/man/man1 0755 bin bin
f none SUNWcadap/man/man1/file4.1=/home/jane/SUNWcadap/man/man1/file4.1 0444 bin bin
f none SUNWcadap/man/man1/file3.1=/home/jane/SUNWcadap/man/man1/file3.1 0444 bin bin
```

Adding Functionality to a `prototype` File

Besides defining every package object in the `prototype` file, you can also:

- Define additional objects to be created at install time.

- Create links at install time.

- Distribute packages over multiple volumes.

- Nest `prototype` files.

- Set a default value for the *mode*, *owner*, and *group* fields.

- Provide a search path for the `pkgmk` command.

- Set environment variables.

See the following sections for information on making these changes.

Defining Additional Objects to Be Created at Install Time

You can use the `prototype` file to define objects that are not actually delivered on the installation medium. During installation, using the `pkgadd` command, these objects are created with the required file types, if they do not already exist at the time of installation.

To specify that an object be created on the target system, add an entry for it in the `prototype` file with the appropriate file type.

For example, if you want a directory created on the target system, but do not want to deliver it on the installation medium, make the following entry for the directory in the `prototype` file:

```
d none /directory 0644 root other
```

If you want to create an empty file on the target system, an entry for the file in the `prototype` file might look like:

```
f none filename=/dev/null 0644 bin bin
```

The only objects that *must* be delivered on the installation medium are regular files and edit scripts (file types e, v, f) and the directories required to contain them. Any additional objects are created without reference to the delivered objects, directories, named pipes, devices, hard links, and symbolic links.

Creating Links at Install Time

To create links during package installation, define the following in the `prototype` file entry for the linked object:

- Its file type as `l` (a link) or `s` (a symbolic link).

- Its path name with the format *path1=path2* where *path1* is the destination and *path2* is the source file. As a general rule, *path2* of a link should never be absolute, but should instead be relative to the directory portion of *path1*. For example, a `prototype` file entry defining a symbolic link could be:

```
s none etc/mount=../usr/etc/mount
```

Relative links would be specified in this manner whether the package is installed as absolute or relocatable.

Distributing Packages Over Multiple Volumes

When you build your package with the `pkgmk` command, it performs the calculations and actions necessary to organize a multiple volume package. A multiple volume package is called a *segmented* package.

However, you can use the optional *part* field in the `prototype` file to define in which part you want an object to be located. A number in this field overrides the `pkgmk` command 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 formatted as file systems. If the volumes are preassigned by the developer, the `pkgmk` command will issue an error if there is insufficient space on any volume.

Nesting `prototype` Files

You can create multiple `prototype` files and then include them, using the `!include` command in the `prototype` file. You might want to do this for easier maintenance.

In the following example there are three `prototype` files, the main one (`prototype`) being edited, and the two (`proto2` and `proto3`) that are being included.

```
!include /source-dir/proto2
!include /source-dir/proto3
```

Setting Default Values for the mode, owner, and group Fields

To set default values for the *mode*, *owner*, and *group* fields for specific package objects, you can insert the `!default` command into the `prototype` file. For example,

```
!default 0644 root other
```

Note - This command's range starts from where it is inserted and extends to the end of the file, but does not span to included files.

However, for directories (file type `d`) and editable files (file type `e`) that you know exist on target systems (like `/usr` or `/etc/vfstab`), make sure that the *mode*, *owner*, and *group* fields in the `prototype` file are set to question marks (`?`). That way you will not destroy existing settings that a site administrator may have modified.

Providing a Search Path for the `pkgmk` Command

If the source location for package objects is different than their destination location, and you do not want to use the *path1=path2* format as described in "A Brief Word on an Object's Source and Destination Locations" on page 24, then you can use the `!search` command in the `prototype` file. For example, if you created a directory, `pkgfiles`, in your home directory, and it contains all of your information files and installation scripts, you can specify that that directory be searched when the package is built with the `pkgmk` command.

The command in the prototype file would look like:

```
!search /home-dir/pkgfiles
```

Note - Search requests do not span to included files. In addition, a search is limited to the specific directories listed, and will *not* search recursively.

Set Environment Variables

You can also add commands to the `prototype` file of the form `!`*PARAM=value*. Commands of this form define variables in the current environment. If you have multiple `prototype` files, the scope of this command is local to the `prototype` file where it is defined.

The variable *PARAM* can begin with either a lowercase or uppercase letter, but its value must be known at build time, or the `pkgmk` command will abort with an error. For more information on the difference between build and install variables, see "Package Environment Variables" on page 12.

▼ How to Create a `prototype` File Using the `pkgproto` Command

Note - It is easier to create information files and installation scripts before creating a `prototype` file. However, this is not required, and you can always edit the `prototype` file after changing your package contents. For more information on information files and installation scripts, see Chapter 3.

1. **Determine which package objects will be absolute and which will be relocatable, if not done already.**

   For information that will help you complete this task, see "The *path* Field" on page 22.

2. **Organize your package's objects to mimic their location on the target system.**

   If you already organized your packages as described in "Organizing a Package's Contents" on page 18, note that you may need to make some changes based on your decisions in Step 1 on page 32. If you have not organized your package yet, you should do so now (otherwise you cannot use the `pkgproto` command to create a basic `prototype` file).

3. **If your package has collectively relocatable objects, edit the** `pkginfo` **file to set the** `BASEDIR` **parameter to the appropriate value.**

   For example:

```
BASEDIR=/opt
```

For information on collectively relocatable objects, see "Collectively Relocatable Objects" on page 23.

4. **If your package has individually relocatable objects, create a** `request` **script to prompt the installer for the appropriate path name or a** `checkinstall` **script to determine the appropriate path from file system data.**

For Information On ...	See ...
Creating a `request` script	"How to Write a `request` Script" on page 55
Creating a `checkinstall` script	"How to Gather File System Data" on page 57
Individually relocatable objects	"Individually Relocatable Objects" on page 23

5. **Change the owner and group on all of your package components to be the intended owner and group on the target systems.**

   Use the `chown -R` and the `chgrp -R` commands on your package directory and information files directory.

6. **Execute the** `pkgproto` **command to create a basic** `prototype` **file.**

   The `pkgproto` command scans your directories to create a basic file. For example:

```
$ cd package-directory
$ pkgproto ./package-directory > prototype
```

   Like the `pkginfo` file, the `prototype` file can be located anywhere on your system. However, it might be a good idea to keep your information files and installation scripts in one place, for easy access and maintenance. For additional information on the `pkgproto` command, see the **pkgproto**(1) man page.

7. **Edit the** `prototype` **file using your favorite text editor, and add entries for files of type** v**,** e**,** x**, and** i**.**

For information on the specific changes you may need to make, see "Fine-Tuning a `prototype` File Created With the `pkgproto` Command" on page 27.

8. **Optional. If you are using multiple classes, edit the** `prototype` **and** `pkginfo` **files using your favorite text editor to make the necessary changes, and create corresponding class action scripts.**

   For information on the specific changes you may need to make, see "Fine-Tuning a `prototype` File Created With the `pkgproto` Command" on page 27 and "Writing Class Action Scripts" on page 61.

9. **Edit the** `prototype` **file using your favorite text editor to redefine path names and change other field settings.**

   For more information, see "Fine-Tuning a `prototype` File Created With the `pkgproto` Command" on page 27.

10. **Optional. Edit the** `prototype` **file using your favorite text editor to add functionality to your** `prototype` **file.**

    For more information, see "Adding Functionality to a `prototype` File" on page 29.

11. **Save your changes and quit the editor.**

Where to Go Next

If you are ready to go to the next task, see "How to Build a Package" on page 35.

Building a Package

Use the `pkgmk` command to build your package. This command takes all of the objects defined in the `prototype` file, puts them into directory format, creates the `pkgmap` file (replacing the `prototype` file), and produces an installable package to be used as input to the `pkgadd` command.

Using the Simplest `pkgmk` Command

The simplest form of this command is the `pkgmk` command itself, without any options. Before using the `pkgmk` command without options, make sure your current working directory contains the package's `prototype` file. The output of the command, files and directories, are written to the `/var/spool/pkg` directory.

The `pkgmap` File

When you build a package with the `pkgmk` command, it creates a `pkgmap` file that replaces the `prototype` file. The `pkgmap` file from the previous example looks like:

```
$ more pkgmap
: 1 3170
1 d none SUNWcadap 0755 root sys
1 d none SUNWcadap/demo 0755 root bin
1 f none SUNWcadap/demo/file1 0555 root bin 14868 45617 837527496
1 d none SUNWcadap/lib 0755 root bin
1 f none SUNWcadap/lib/file2 0644 root bin 1551792 62372 837527499
1 d none SUNWcadap/man 0755 bin bin
1 d none SUNWcadap/man/man1 0755 bin bin
1 f none SUNWcadap/man/man1/file3.1 0444 bin bin 3700 42989 837527500
1 f none SUNWcadap/man/man1/file4.1 0444 bin bin 1338 44010 837527499
1 f none SUNWcadap/man/windex 0644 root other 157 13275 837527499
1 d none SUNWcadap/srcfiles 0755 root bin
1 f none SUNWcadap/srcfiles/file5 0555 root bin 12208 20280 837527497
1 f none SUNWcadap/srcfiles/file6 0555 root bin 12256 63236 837527497
1 i pkginfo 140 10941 837531104
$
```

The format of this file is very similar to that of the `prototype` file. However, the `pkgmap` file includes the following information.

- The first line indicates the number of volumes that the package spans, and the approximate size the package will be when installed.

  For example, `: 1 3170` indicates that the package spans one volume and will use approximately 3170 512-byte blocks when it is installed.

- There are three additional fields defining the size, checksum, and modification time for each package object.

- The package objects are listed in alphabetical order by class and by path name to enhance the time it takes to install the package.

▼ How to Build a Package

1. **Create a** `pkginfo` **file, if not done already.**

   For step-by-step instructions, see "How to Create a `pkginfo` File" on page 17.

2. **Create a** `prototype` **file, if not done already.**

   For step-by-step instructions, see "How to Create a `prototype` File Using the `pkgproto` Command" on page 32.

3. **Make your current working directory the same directory that contains your package's** `prototype` **file.**

4. **Build the package.**

```
$ pkgmk [-o] [-a arch] [-b base-src-dir] [-d device]
    [-f filename] [-l limit] [-p pstamp] [-r rootpath]
    [-v version] [PARAM=value] [pkginst]
```

−o	Overwrites the existing version of the package.
−a *arch*	Overrides the architecture information in the pkginfo file.
−b *base-src-dir*	Requests that *base-src-dir* be added to the beginning of relocatable path names when the pkgmk command is searching for objects on the development system.
−d *device*	Specifies that the package should be copied onto *device*, which may be a an absolute directory path name, diskette, or removable disk.
−f *filename*	Names a file, *filename*, that is used as your prototype file. The default names are prototype or Prototype.
−l *limit*	Specifies the maximum size, in 512-byte blocks, of the output device.
−p *pstamp*	Overrides the production stamp definition in the pkginfo file.
−r *rootpath*	Requests that the root directory *rootpath* be used to locate objects on the development system.
−v *version*	Overrides the version information in the pkginfo file.
PARAM=value	Sets global environment variables. Variables beginning with lowercase letters are resolved at build time. Those beginning with uppercase letters are placed into the pkginfo file for use at install time.
pkginst	Specifies a package by its package abbreviation or a specific instance (for example, SUNWcadap.4)

For more information, see **pkgmk**(1).

5. Verify the contents of the package.

```
$ pkgchk -d pkg-dir pkg-abbrev
Checking uninstalled directory format package pkg-abbrev

from pkg-dir
## Checking control scripts.
## Checking package objects.
## Checking is complete.
$
```

−d *device-name*	Specifies the location of the package. Note that *device-name* can be a full directory path name or the identifiers for a tape, floppy disk, or removable disk.
pkgid	Is the name of one or more packages (separated by spaces) to be checked. If omitted, pkgchk checks all available packages.

The pkgchk command prints what aspects of the package are being checked and displays warnings or errors, as appropriate. For more information on the pkgchk command, see "Verifying the Integrity of a Package" on page 75.

Note - Errors should be considered very seriously; it may mean that a script needs fixing. However, a warning may just be a comment on the style of a script. Check it and move on if you disagree with the output from the pkgchk command.

Where to Go Next

If you want to add any optional information files and installation scripts to your package, see Chapter 3. Otherwise, after you build the package, you should verify its integrity. Chapter 4 explains how to do this, and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Example—Building a Package

This example uses the prototype file created in "Fine-Tuning a prototype File Created With the pkgproto Command" on page 27.

```
$ cd /home/jane/InfoFiles
$ pkgmk
## Building pkgmap from package prototype file.
## Processing pkginfo file.
WARNING: parameter  set to "system990708093144"
WARNING: parameter  set to "none"
```

```
## Attempting to volumize 13 entries in pkgmap.
part  1 -- 3170 blocks, 17 entries
## Packaging one part.
/var/spool/pkg/SUNWcadap/pkgmap
/var/spool/pkg/SUNWcadap/pkginfo
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/demo/file1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/lib/file2
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file3.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file4.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/windex
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file5
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file6
## Validating control scripts.
## Packaging complete.
$
```

Example—Specifying a Source Directory for Relocatable Files

If your package contains relocatable files, you can use the −b *base-src-dir* option to the
pkgmk command to specify a path name to be added to the beginning of the
relocatable path names while the package is being created. This is useful if you
haven't used the *path1=path2* format for relocatable files or specified a search path
with the !search command in the prototype file.

For example, to build a package using the sample prototype file created by the
pkgproto command (see "Creating a prototype File With the pkgproto
Command" on page 26), without modifying the *path* fields, and just adding an entry
for the pkginfo file, the pkgmk command is:

```
$ cd /home/jane/InfoFiles
$ pkgmk -o -b /home/jane
## Building pkgmap from package prototype file.
## Processing pkginfo file.
WARNING: parameter  set to "system960716102636"
WARNING: parameter   set to "none"
## Attempting to volumize 13 entries in pkgmap.
part  1 -- 3170 blocks, 17 entries
## Packaging one part.
/var/spool/pkg/SUNWcadap/pkgmap
/var/spool/pkg/SUNWcadap/pkginfo
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/demo/file1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/lib/file2
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file3.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/man1/file4.1
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/man/windex
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file5
/var/spool/pkg/SUNWcadap/reloc/SUNWcadap/srcfiles/file6
## Validating control scripts.
```

(continued)

```
## Packaging complete.
```

In this example, the package is built in the default directory, /var/spool/pkg, by specifying the −o option (to overwrite the package we created in "Example—Building a Package" on page 37).

Example—Specifying Different Source Directories for Information Files and Package Objects

If you put package information files (such as pkginfo and prototype) and the package objects in two different directories, you can create your package by using the −b *base-src-dir* and −r *rootpath* options to the pkgmk command. If you have your package objects in a directory called /product/pkgbin and the other package information files in a directory called /product/pkgsrc, you could use the following command to place the package in the /var/spool/pkg directory:

```
$ pkgmk -b /product/pkgbin -r /product/pkgsrc -f /product/pkgsrc/prototype
```

Optionally, you could use this command to do the same:

```
$ cd /product/pkgsrc
$ pkgmk -o -b /product/pkgbin
```

In this example, the pkgmk command uses the current working directory to find the remaining parts of the package (like the prototype and pkginfo information files).

Enhancing the Functionality of a Package

This chapter describes how to create optional information files and installation scripts for a package. While Chapter 2 discussed the minimum requirements for making a package, this chapter discusses additional functionality that you can build into a package, based on the criteria you considered when planning how to design your package (for more information, see "Things to Think About Before Building a Package" on page 5).

This is a list of the overview information in this chapter.

■ "Creating Information Files and Installation Scripts Task Map" on page 41

■ "Creating Information Files" on page 43

■ "Creating Installation Scripts" on page 49

Creating Information Files and Installation Scripts Task Map

The table below lists and describes the optional features you can build into a package.

TABLE 3–1   Creating Information Files and Installation Scripts Task Map

Task	Description	For Instructions, Go To ...
1. Create Information Files	*Define Package Dependencies* A definition of package dependencies allows you to specify whether your package is compatible with previous versions, dependent on other packages, or whether other packages are dependent on yours.	"How to Define Package Dependencies" on page 43
	Provide a Copyright Message A `copyright` file provides legal protection for your software application.	"How to Write a Copyright Message" on page 46
	Create Additional Space on the Target System. A `space` file sets aside blocks on the target system, which enables you to create files during installation that are not defined in the `pkgmap` file.	"How to Reserve Additional Space on a Target System" on page 48
2. Create Installation Scripts	*Obtain Information From the Installer* A `request` script enables you to obtain information from the person installing your package.	"How to Write a `request` Script" on page 55
	Gather File System Data Needed For Installation A `checkinstall` script enables you to perform an analysis of the target system and set up the correct environment for, or cleanly halt, the installation.	"How to Gather File System Data" on page 57
	Write Procedure Scripts Enables you to provide customized installation instructions during specific phases of the installation or removal process.	"How to Write Procedure Scripts" on page 60
	Write Class Action Scripts Enables you to specify a set of instructions to be executed during package installation and removal on specific groups of package objects.	"How to Write Class Action Scripts" on page 68

Creating Information Files

This section discusses optional package information files. With these files you can define package dependencies, provide a copyright message, and reserve additional space on a target system.

Defining Package Dependencies

You need to determine whether your package has dependencies on other packages and if any other packages depend on yours. Package dependencies and incompatibilities can be defined with two of the optional package information files, `compver` and `depend`. Delivering a `compver` file lets you name previous 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 - Use the reverse dependency type only when a package that cannot deliver a `depend` file relies on your package.

- An *incompatible package* – meaning your package is incompatible with the named package

The `depend` file resolves only very basic dependencies. If your package depends upon a specific file or its contents or behavior, the `depend` file does not supply adequate precision. In this case, a `request` script or the `checkinstall` script should be used for detailed dependency checking. The `checkinstall` script is also the only script capable of cleanly halting the package installation process.

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).

Refer to **depend**(4) and **compver**(4) for more information.

▼ How to Define Package Dependencies

1. **Make the directory containing your information files the current working directory.**

2. **If previous versions of your package exist and you need to specify that your new package is compatible with them, create a file named** `compver` **with your favorite text editor.**

   List the versions with which your package is compatible, using this format:

string string . . .

string	Is identical to the value assigned to the `VERSION` parameter in the `pkginfo` file, for each compatible package.

3. **Save your changes and quit the editor.**

4. **If your package depends on the existence of other packages, other packages depend on the existence of your package, or your package is incompatible with another package, create a file named** `depend` **with your favorite text editor.**

   Add an entry for each dependency, using this format:

type pkg-abbrev pkg-name
        ⟨*arch*⟩   *version*
        ⟨*arch*⟩   *version* . . .

type	Defines the dependency type. Must be one of the following characters: `P` (prerequisite package), `I` (incompatible package), or `R` (reverse dependency).
pkg-abbrev	Specifies the package abbreviation, such as `SUNWcadap`.
pkg-name	Sepcifies the full package name, such as `Chip designers need CAD application software to design abc chips. Runs only on xyz hardware and is installed in the usr partition.`

(arch)	Optional. Specifies the type of hardware on which the package runs. For example, `sparc` or `x86`. If you specify an architecture, you must use the parentheses as delimiters.
version	Optional. Specifies the value assigned to the VERSION parameter in the `pkginfo` file.

For more information, see **depend**(4).

5. **Save your changes and quit the editor.**

6. **Complete *one* of the following tasks:**
   - If you want to create additional information files and installation scripts, skip to the next task, "How to Write a Copyright Message" on page 46.
   - If you have *not* created your `prototype` file, complete the procedure "How to Create a `prototype` File Using the `pkgproto` Command" on page 32, and skip to Step 7 on page 45.
   - If you have already created your `prototype` file, edit it and add an entry for each file you just created.

7. **Build your package.**
   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Example—`compver` File

In this example, there are four versions of a package: 1.0, 1.1, 2.0, and the new package, 3.0, which is compatible with all the three previous versions. The `compver` file for the newest version might look like:

```
release 3.0
release 2.0
version 1.1
1.0
```

Note - The entries do not have to be in sequential order. However, they should exactly match the definition of the VERSION parameter in each package's pkginfo file. In this example, the package designers used different formats in the first three versions.

Example—depend File

This example assumes that the sample package, SUNWcadap, requires that the SUNWcsr and SUNWcsu packages already be installed on a target system. The depend file for SUNWcadap looks like:

```
P SUNWcsr Core Solaris, (Root)
P SUNWcsu Core Solaris, (Usr)
```

Writing a Copyright Message

You need to decide whether your package should display a copyright message while it is being installed. If so, create the copyright file.

Note - You should include a copyright file to provide legal protection for your software application. It might be a good idea to check with the legal department of your company for the exact wording of the message.

To deliver a copyright message, you must create a file named copyright. During installation, the message is displayed exactly as it appears in the file (with no formatting). See **copyright**(4) for more information.

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

▼ How to Write a Copyright Message

1. **Make the directory containing your information files the current working directory.**

2. **Create a file named** copyright **with your favorite text editor.**
   Enter the text of the copyright message exactly as you want it to appear as your package is installed.

3. **Save your changes and quit the editor.**

4. **Complete *one* of the following tasks.**

   - If you want to create additional information files and installation scripts, skip to the next task, "How to Reserve Additional Space on a Target System" on page 48.

   - If you have *not* created your `prototype` file, complete the procedure "How to Create a `prototype` File Using the `pkgproto` Command" on page 32, and skip to Step 5 on page 47.

   - If you have already created your `prototype` file, edit it and add an entry for the information file you just created.

5. **Build your package.**

   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Example—`copyright` File

For example, a partial copyright message might look like:

```
Copyright (c) 1999 Company Name
All Rights Reserved

This product is protected by copyright and distributed under
licenses restricting copying, distribution and decompilation.
```

Reserving Additional Space on a Target System

You need to determine whether your package needs additional disk space on the target system (space in addition to that required by the package objects). If so, create the `space` information file. This is different than creating empty files and directories at installation time as discussed in "Defining Additional Objects to Be Created at Install Time" on page 30.

While the `pkgadd` command ensures that there is enough disk space to install your package based on the object definitions in the `pkgmap` file, a package may require additional disk space beyond that needed by the objects defined in the `pkgmap` file. For example, your package might create a file after installation, which may contain a database, log files, or some other growing file that consumes disk space. To be sure that there is space set aside for it, you should include a `space` file specifying the

disk space requirements. The pkgadd command checks for the additional space specified in a space file. Refer to **space**(4) for more information.

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

▼ How to Reserve Additional Space on a Target System

1. **Make the directory containing your information files the current working directory.**

2. **Create a file named** space **with your favorite text editor.**

   Enter any additional disk space requirements needed by your package, using this format:

pathname   *blocks*   *inodes*

pathname	Specifies a directory name, which may or may not be the mount point for a file system.
blocks	Specifies the number of 512-byte blocks that you want reserved.
inodes	Specifies the number of required inodes.

   For more information, see **space**(4).

3. **Save your changes and quit the editor.**

4. **Complete *one* of the following tasks.**
   - If you want to create installation scripts, skip to the next task, "How to Write a request Script" on page 55.
   - If you have *not* created your prototype file, complete the procedure in "How to Create a prototype File Using the pkgproto Command" on page 32, and skip to Step 5 on page 48.
   - If you have already created your prototype file, edit it and add an entry for the information file you just created.

5. **Build your package.**

   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Example—`space` File

This example `space` file specifies that 1000 512-byte blocks and 1 inode be set aside in `/opt` on the target system.

```
/opt    1000    1
```

Creating Installation Scripts

This section discusses optional package installation scripts. The `pkgadd` command automatically performs all the actions necessary to install a package using the package information files as input. You do *not* have to supply any package installation scripts. However, if you want to create customized installation procedures for your package, you can do so with installation scripts. Installation scripts:

- Must be executable by the Bourne shell (`sh`).
- Must contain Bourne shell commands and text.
- Do not need to contain the `#!/bin/sh` shell identifier.
- Need not be an executable file.

There are four types of installation scripts with which you can perform customized actions:

- The `request` script

  The `request` script solicits data from the administrator installing a package for assigning or redefining environment variables.

- The `checkinstall` script

  The `checkinstall` script examines the target system for needed data, can set or modify package environment variables, and determines whether or not the installation proceeds.

Note - The `checkinstall` script is available starting with the Solaris 2.5 and compatible releases.

- Procedure scripts

Procedure scripts identify 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`.

- Class action scripts

  Class action scripts define an action or set of actions that should be applied to a class of files during installation or removal. You can define your own classes or use one of the four standard classes (`sed`, `awk`, `build`, and `preserve`).

Script Processing During Package Installation

The type of scripts you use depends on when the action of the script is needed during the installation process. As a package is installed, the `pkgadd` command performs the following steps:

1. Executes the `request` script.

   This is the *only* point at which your package can solicit input from the administrator installing the package.

2. Executes the `checkinstall` script.

   The `checkinstall` script gathers file system data and can create or alter environment variable definitions to control the subsequent installation. For more information on package environment variables, see "Package Environment Variables" on page 12.

3. Executes the `preinstall` script.

4. Installs package objects, for each class to be installed.

   Installation of these files occurs class by class, and class action scripts are executed accordingly. The list of classes operated on and the order in which they should be installed is initially defined with the CLASSES parameter in your `pkginfo` file. However, your `request` script or `checkinstall` script can change the value of the CLASSES parameter. For more information on how classes are processed during installation, see "How Classes Are Processed During Installation" on page 61.

   a. Creates symbolic links, devices, named pipes, and required directories.

   b. Installs the regular files (file types `e`, `v`, `f`), based on their class.

      The class action script is passed only regular files to install. All other package objects are created automatically from information in the `pkgmap` file.

   c. Creates all hard links.

5. Executes the `postinstall` script.

Script Processing During Package Removal

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

1. Executes the `preremove` script.
2. Removes the package objects, for each class.

   Removal also occurs class by class. Removal scripts are processed in the reverse order of installation, based on the sequence defined in the `CLASSES` parameter. For more information on how classes are processed during installation, see "How Classes Are Processed During Installation" on page 61.

   a. Removes hard links.

   b. Removes regular files.

   c. Removes symbolic links, devices, and named pipes.
3. Executes the `postremove` script.

The `request` script is not processed at the time of package removal. However, its output (a list of environment variables) is retained in the installed package and made available to removal scripts.

Package Environment Variables Available to Scripts

The following groups of environment variables are available to all installation scripts. Some of the environment variables can be modified by a `request` or `checkinstall` script.

- The `request` or `checkinstall` script can set or modify any of the standard parameters in the `pkginfo` file except for the required parameters. The standard installation parameters are described in detail in **pkginfo**(4).

Note - The `BASEDIR` parameter can only be modified starting with the Solaris 2.5 and compatible releases.

- You can define your own installation environment variables by assigning values to them in the `pkginfo` file. Such environment variables must be alphanumeric with initial capital letters. Any of these environment variables can be changed by a `request` or `checkinstall` script.

- Both a `request` script and a `checkinstall` script can define new environment variables by assigning values to them and putting them in the installation environment.

- The table below lists environment variables that are available to all installation scripts through the environment. None of these can be modified by a script.

TABLE 3–2 Package Environment Variables Available to Scripts

Environment Variable	Description
CLIENT_BASEDIR	The base directory with respect to the target system. While BASEDIR is the variable to use if you are referring to a specific package object from the install system (most likely a server), CLIENT_BASEDIR is the path to include in files placed on the client system. CLIENT_BASEDIR exists if BASEDIR exists and is identical to BASEDIR if there is no PKG_INSTALL_ROOT.
INST_DATADIR	The directory where the package now being read is located. If the package is being read from a tape, this will be the location of a temporary directory where the package has been transferred into directory format. In other words, assuming there is no extension to the package name (for example, SUNWstuff.d), the request script for the current package would be found at $INST_DATADIR/$PKG/install.
PATH	The search list used by sh to find commands on script invocation. PATH is usually set to /sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin
PKGINST	The instance identifier of the package being installed. If another instance of the package is not already installed, the value is the package abbreviation (for example, SUNWcadap). Otherwise, it is the package abbreviation followed by a suffix, such as SUNWcadap.4.
PKGSAV	The directory where files can be saved for use by removal scripts or where previously saved files can be found. Available only in the Solaris 2.5 and compatible releases.
PKG_INSTALL_ROOT	The root file system on the target system where the package is being installed. It exists only if the pkgadd and pkgrm commands were invoked with the −R option. This conditional existence facilitates its use in procedure scripts in the form ${PKG_INSTALL_ROOT}/*somepath*.
PKG_NO_UNIFIED	Is an environment variable that gets set if the pkgadd and pkgrm commands were invoked with the −M and −R options. This environment variable is passed to any package installation script or package command that is part of the package environment.
UPDATE	This environment variable does not exist under most installation environments. If it does exist (with the value yes), it means that a package with the same name, version, and architecture is already installed on the system or that this package is overwriting an installed package of the same name at the direction of the administrator. In these events, the original base directory is *always* used.

Obtaining Package Information for a Script

Two commands can be used from 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 requested environment variables.

  See **pkginfo**(1), **pkgparam**(1), and Chapter 4 for more information.

Exit Codes for Scripts

Each script must exit with one of the exit codes shown in the following table.

TABLE 3–3   Installation Script Exit Codes

Code	Meaning
0	Successful completion of script.
1	Fatal error. Installation process is terminated at this point.
2	Warning or possible error condition. Installation continues. A warning message is displayed at the time of completion.
3	The `pkgadd` command is cleanly halted. Only the `checkinstall` script returns this code,
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	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 Chapter 5 for examples of exit codes returned by installation scripts.

Note - All installation scripts delivered with your package should have an entry in the `prototype` file. The file type should be `i` (for package installation script).

Writing a `request` Script

The `request` script is the only way your package can interact directly with the administrator installing it. It can be used, for example, to ask the administrator if optional pieces of a package should be installed.

The output of a `request` script must be a list of environment variables and their values. This list can include any of the parameters you created in the `pkginfo` file and the `CLASSES` and `BASEDIR` parameters. The list can also introduce environment variables that have not been defined elsewhere (although the `pkginfo` file should always provide default values, when practical). For more information on package environment variables, see "Package Environment Variables" on page 12.

When your `request` script assigns values to a environment variable, it must then make those values available to the `pkgadd` command and other package scripts.

`request` Script Behaviors

- The `request` script cannot modify any files. It only interacts with administrators installing the package and creates a list of environment variable assignments based upon that interaction. To enforce this restriction, the `request` script is executed as the non-privileged user `install` if that user exists; otherwise it is executed as the non-privileged user `nobody`. The `request` script does not have superuser authority.

- The `pkgadd` command calls the `request` script with one argument that names the script's response file (the file that stores the administrator's responses).

- The `request` script is not executed during package removal. However, the environment variables assigned by the script are saved and are available during removal.

Design Rules for `request` Scripts

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

- The environment variable assignments should be added to the installation environment for use by the `pkgadd` command and other packaging scripts by writing them to the response file (known to the script as `$1`).

- System environment variables and standard installation environment variables, except for the `CLASSES` and `BASEDIR` parameters, cannot be modified by a `request` script. Any of the other environment variables you created can be changed.

Note - A `request` script can only modify the `BASEDIR` parameter starting with the Solaris 2.5 and compatible releases.

- Every environment variable that the `request` script may manipulate should be assigned a default value in the `pkginfo` file.

- The format of the output list should be *PARAM=value*. For example:

```
CLASSES=none class1
```

- The administrator's terminal is defined as standard input to the `request` script.

- Do not perform any special analysis of the target system in a `request` script. It is risky to test the system for the presence of certain binaries or behaviors, and set environment variables based upon that analysis, because there is no guarantee that the `request` script will actually be executed at install time. The administrator installing the package may provide a response file that will insert the environment variables without ever calling the `request` script. If the `request` script is also evaluating the target file system, that evaluation may not happen. An analysis of the target system for special treatment is best left to the `checkinstall` script.

Note - If it is possible that the administrators who will be installing your package will be using the JumpStart™ product, then the installation of your package must *not* be interactive. This implies that either you should not provide a `request` script with your package, or you need to communicate to the administrator that they should use the `pkgask` command, prior to installation, to store their responses to the `request` script. For more information on the `pkgask` command, see **pkgask**(1M).

▼ How to Write a `request` Script

1. **Make the directory containing your information files the current working directory.**

2. **Create a file named** `request` **with your favorite text editor.**

3. **Save your changes and quit the editor when you are done.**

4. **Complete *one* of the following tasks.**
   - If you want to create additional installation scripts, skip to the next task, "How to Gather File System Data" on page 57.
   - If you have *not* created your `prototype` file, complete the procedure "How to Create a `prototype` File Using the `pkgproto` Command" on page 32, and skip to Step 5 on page 55.
   - If you have already created your `prototype` file, edit it and add an entry for the installation script you just created.

5. **Build your package.**
   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Example—Writing a `request` Script

When a `request` script assigns values to environment variables, it must make those values available to the `pkgadd` command. This example shows a `request` script segment that performs this task for the four environment variables CLASSES, NCMPBIN, EMACS, and NCMPMAN. (These were defined in an interactive session with the administrator earlier in the script.)

```
# make environment variables available to installation
# service and any other packaging script we might have

cat >$1 <<!
CLASSES=$CLASSES
NCMPBIN=$NCMPBI
EMACS=$EMACS
NCMPMAN=$NCMPMAN
!
```

Gathering Data With the `checkinstall` Script

The `checkinstall` script is executed shortly after the optional `request` script. It runs as the user `install`, if such a user exists, or the user `nobody`. The `checkinstall` script does not have the authority to change file system data; it is strictly a data gatherer. However, based on the information it gathers, it can create or modify environment variables in order to control the course of the resulting installation. It is also capable of cleanly halting the installation process.

The `checkinstall` script is intended to perform basic checks on a file system that would not be normal for the `pkgadd` command. For example, it can be used to check ahead to determine if any files from the current package are going to overwrite existing files, or manage general software dependencies (the `depend` file only manages package-level dependencies).

Unlike the `request` script, the `checkinstall` script is executed whether or not a response file is provided; and, its presence does not brand the package as "interactive." This means that the `checkinstall` script can be used in situations where a `request` script is forbidden or administrative interaction is not practical.

Note - The `checkinstall` script is available starting with the Solaris 2.5 and compatible releases.

`checkinstall` Script Behaviors

- The `checkinstall` script cannot modify any files. It only analyzes the state of the system and creates a list of environment variable assignments based upon that interaction. To enforce this restriction, the `checkinstall` script is executed as the non-privileged user `install` if that user exists; otherwise it is executed as the non-privileged user `nobody`. The `checkinstall` script does not have superuser authority.

- The `pkgadd` command calls the `checkinstall` script with one argument that names the script's response file (the file that stores the administrator's responses).

- The `checkinstall` script is not executed during package removal. However, the environment variables assigned by the script are saved and are available during removal.

Design Rules for `checkinstall` Scripts

- There can be only one `checkinstall` script per package and it must be named `checkinstall`.

- The environment variable assignments should be added to the installation environment for use by the `pkgadd` command and other packaging scripts by writing them to the response file (known to the script as `$1`).

- System environment variables and standard installation environment variables, except for the `CLASSES` and `BASEDIR` parameters, cannot be modified by a `checkinstall` script. Any of the other environment variables you created can be changed.

- Every environment variable that the `checkinstall` script may manipulate should be assigned a default value in the `pkginfo` file.

- The format of the output list should be *PARAM=value*. For example:

```
CLASSES=none class1
```

- Administrator interaction is not permitted during execution of a `checkinstall` script. All administrator interaction is restricted to the `request` script.

▼ How to Gather File System Data

1. **Make the directory containing your information files the current working directory.**

2. **Create a file named `checkinstall` with your favorite text editor.**

3. **Save your changes and quit the editor when you are done.**

4. **Complete *one* of the following tasks.**

- If you want to create additional installation scripts, skip to the next task, "How to Write Procedure Scripts" on page 60.

- If you have *not* created your `prototype` file, complete the procedure "How to Create a `prototype` File Using the `pkgproto` Command" on page 32, and skip to Step 5 on page 58.

- If you have already created your `prototype` file, edit it and add an entry for the installation script you just created.

5. **Build your package.**

   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Example—Writing a `checkinstall` Script

This example `checkinstall` script checks to see if database software needed by the `SUNWcadap` package is installed.

```
# checkinstall script for SUNWcadap
#
# This confirms the existence of the required specU database

# First find which database package has been installed.
pkginfo -q SUNWspcdA # try the older one

if [ $? -ne 0 ]; then
   pkginfo -q SUNWspcdB # now the latest

   if [ $? -ne 0 ]; then # oops
      echo "No database package can be found. Please install the"
      echo "SpecU database package and try this installation again."
      exit 3  # Suspend
   else
      DBBASE="`pkgparam SUNWsbcdB BASEDIR`/db" # new DB software
   fi
else
   DBBASE="`pkgparam SUNWspcdA BASEDIR`/db" # old DB software
fi

# Now look for the database file we will need for this installation
if [ $DBBASE/specUlatte ]; then
   exit 0  # all OK
else
   echo "No database file can be found. Please create the database"
```

(continued)

```
   echo "using your installed specU software and try this"
   echo "installation again."
   exit 3  # Suspend
fi
```

Writing Procedure Scripts

The procedure scripts provide a set of instructions to be performed at particular points in package installation or removal. The four procedure scripts must be named one of the predefined names, depending on when the instructions are to be executed, and are executed without arguments.

- The `preinstall` script

  Runs before class installation begins. No files should be installed by this script.

- The `postinstall` script

  Runs after all volumes have been installed.

- The `preremove` script

  Runs before class removal begins. No files should be removed by this script.

- The `postremove` script

  Runs after all classes have been removed.

Procedure Script Behaviors

Procedure scripts are executed as `uid=root` and `gid=other`.

Design Rules for Procedure Scripts

- Each script should be able to be executed more than once since it is executed once for each volume in a package. This means that executing a script any number of times with the same input produces the same results as executing the script only once.

- Each procedure script that installs a package object not in the `pkgmap` file must use the `installf` command to notify the package database that it is adding or modifying a path name. After all additions or modifications are complete, this command should be invoked with the `−f` option. Only `postinstall` and `postremove` scripts may install package objects in this way. See **installf**(1M) and Chapter 5 for more information.

- Administrator interaction is not permitted during execution of a procedure script. All administrator interaction is restricted to the `request` script.

- Each procedure script that removes files not installed from the `pkgmap` file must use the `removef` command to notify the package database that it is removing a path name. After removal is complete, this command should be invoked with the −f option. See **removef**(1M) and Chapter 5 for details and examples.

Note - The `installf` and `removef` commands must be used because procedure scripts are not automatically associated with any path names listed in the `pkgmap` file.

▼ How to Write Procedure Scripts

1. **Make the directory containing your information files the current working directory.**

2. **Create one or more procedure scripts with your favorite text editor.**

   A procedure script must be named one of the predefined names: `preinstall`, `postinstall`, `preremove`, or `postremove`.

3. **Save your changes and quit the editor.**

4. **Complete *one* of the following tasks.**
   - If you want to create class action scripts, skip to the next task, "How to Write Class Action Scripts" on page 68.
   - If you have *not* created your `prototype` file, complete the procedure "How to Create a `prototype` File Using the `pkgproto` Command" on page 32, and skip to Step 5 on page 60.
   - If you have already created your `prototype` file, edit it and add an entry for each installation script you just created.

5. **Build your package.**

   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Writing Class Action Scripts

Defining Object Classes

Object classes allow a series of actions to be performed on a group of package objects at installation or removal. You assign objects to a class in the `prototype` file. All package objects must be given a class, although the class of `none` is used by default 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).

Note - Objects defined in the `pkgmap` file that belong to a class not listed in this parameter in the `pkginfo` file will *not* be installed.

The CLASSES list determines the order of installation. Class `none` is always installed first, if present, and removed last. Since directories are the fundamental support structure for all other file system objects, they should all be assigned to the `none` class. Exceptions can be made, but as a general rule, the `none` class is safest. The reason for this is to ensure that the directories are created before the objects they will contain and also to ensure that no attempt is made to delete a directory before it has been emptied.

How Classes Are Processed During Installation

The following list describes 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. The `pkgadd` command creates a path name list.

   The `pkgadd` command creates a list of path names upon which the action script will operate. Each line of this list contains source and destination path names, separated by a space. The source path name indicates where the object to be installed resides on the installation volume and the destination path name indicates the location on the target system where the object should be installed. The contents of the list are restricted by the following criteria:

   - The list contains only path names belonging to the associated class.

   - If the attempt to create the package object fails, directories, named pipes, character devices, block devices, and symbolic links are included in the list with the source path name set to `/dev/null`. Normally they will be automatically created by the `pkgadd` command (if not already in existence) and given proper attributes (mode, owner, group) as defined in the `pkgmap` file.

   - Linked files where the file type is `l` are not included in the list under any circumstances. Hard links in the given class are created in item 4.

2. If no class action script is provided for installation of a particular class, the path names in the generated list are 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 item 1. If this is the last volume of the package, or there are no more objects in this class, the script is executed with the single argument of ENDOFCLASS.

Note - Even if there are no regular files of this class anywhere in the package, the class action script will be called at least once with an empty list and the ENDOFCLASS argument.

---

4. The pkgadd command performs a content and attribute audit and creates hard links.

   After successfully executing items 2 or 3, the pkgadd command audits both content and attribute information for the list of path names. The pkgadd command creates the links associated with the class automatically. Detected attribute inconsistencies are corrected for all path names in the generated list.

How Classes Are Processed During Removal

Objects are removed class by class. Classes that exist for a package but that are not listed in the CLASSES parameter are removed first (for example, an object installed with the installf command). Classes listed in the CLASSES parameter are removed in reverse order. The none class is always removed last. The following list describes the system actions that occur when a class is removed:

1. The pkgrm command creates a path name list.

   The pkgrm command creates a list of installed path names that belong to the indicated class. Path names referenced by another package are excluded from the list unless their file type is e (meaning the file should be edited upon installation or removal).

   If the package being removed modified any files of type e during installation, it should remove just the lines it added. Do not delete a non-empty editable file; just remove the lines the package added.

2. If there is no class action script, the path names are deleted.

   If your package has no removal class action script for the class, all the path names in the list generated by the pkgrm command are deleted.

Note - Files with a file type of e (editable), which are not assigned to a class and an associated class action script, will be removed at this point, even if the path name is shared with other packages.

---

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

The `pkgrm` command invokes the class action script with standard input for the script containing the list generated in item 1.

4. The `pkgrm` command performs an audit.

   After successfully executing the class action script, the `pkgrm` command removes knowledge of the path names from the package database unless a path name is referenced by another package.

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 path names based on their class definition. (See Chapter 5 for examples of class action scripts.)

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

Name Format	Description
`i.`*class*	Operates on path names in the indicated class during package installation.
`r.`*class*	Operates on path names in the indicated class during package removal.

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

Note - This file name format is not used for files belonging to the `sed`, `awk`, or `build` system classes. For more information on these special classes, see "The Special System Classes" on page 64.

Class Action Script Behaviors

- Class action scripts are executed as `uid=root` and `gid=other`.

- A script is executed for all files in the given class on the current volume (for example, floppy disk).

- The `pkgadd` and `pkgrm` commands create a list of all objects listed in the `pkgmap` file that belong to the class. As a result, a class action script can act only upon path names defined in the `pkgmap` that belong to a particular class.

- When a class action script is executed for the last time (that is, no more files are found belonging to that class), the class action script will be executed once with the keyword argument `ENDOFCLASS`.

- Administrator interaction is not permitted during execution of a class action script.

Design Rules for Class Action Scripts

- If a package spans more than one volume, the class action script is executed once for each volume that contains at least one file belonging to a class. Consequently, each script must be able to be executed more than once. 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.

- When a file is part of a class that has a class action script, the script must install the file. The `pkgadd` command does not install files for which a class action script exists, although it does verify the installation.

- A class action script should never add, remove, or modify a path name or system attribute that does not appear in the list generated by the `pkgadd` command. For more information on this list, see item 1 in "How Classes Are Processed During Installation" on page 61.

- When your script sees the `ENDOFCLASS` argument, put post-processing actions (like clean up) into your script.

- All administrator interaction is restricted to the `request` script. Do not try to obtain information from the administrator using a class action script.

The Special System Classes

The system provides four special classes. They are:

- 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 dynamically construct or modify a file using Bourne shell commands.

- The `preserve` class

  Provides a method to preserve files that should not be overwritten by future package installations.

If there are several files in a package that require special processing that can be fully defined through `sed`, `awk`, or `sh` commands, installation will be faster by using the system classes rather than multiple classes and their corresponding class action scripts.

The `sed` *Class Script*

The `sed` class provides a method to modify an existing object on the target system. The `sed` class action script executes automatically at installation if a file belonging to class `sed` exists. The name of the `sed` class action script should be the same as the name of the file on which the instructions will be executed.

A `sed` class action script delivers `sed` instructions in the format shown in the figure below.

```
# comment, which may appear on any line in the file

!install
# sed(1) instructions which will be invoked during
# installation of the object

[address [,address]] function [arguments]

   . . .

!remove

# sed(1) instructions to be invoked during the removal process

[address [,address]] function [arguments]
```

Figure 3–1   Format of a `sed` Class Action Script

Two commands indicate when instructions should be executed. The `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 which order these commands are used in the file.

For more information on `sed` instructions, see **sed**(1). For examples of `sed` class action scripts, see Chapter 5.

The `awk` *Class Script*

The `awk` class provides a method to modify an existing object on the target system. Modifications are delivered as `awk` instructions in an `awk` class action script.

The `awk` class action script is executed automatically at installation if a file belonging to class `awk` exists. Such a file contains instructions for the `awk` class script in the format shown in the figure below.

```
# 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)
```

Figure 3–2   Format of an `awk` Class Action Script

Two commands indicate when instructions should be executed. The `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 these commands are used in the file.

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

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

For more information on `awk` instructions, see **awk**(1).

The `build` *Class Script*

The `build` class creates or modifies a package object file by executing Bourne shell
instructions. These instructions are delivered as the package object, which runs
automatically at installation if it belongs to the `build` class.

The name of the `build` class action script should be the same as the name of the file
on which the instructions will be executed, and must be executable by the `sh`
command. The script's output becomes the new version of the file as it is built or
modified.

For example, if a package delivers a default file, `/etc/randomtable`, and if the file
does not already exist on the target system, the `prototype` file entry might be:

```
e build /etc/randomtable ? ? ?
```

and the package object, `/etc/randomtable`, might look like this:

```
!install
# randomtable builder
if [ -f $PKG_INSTALL_ROOT/etc/randomtable ]; then
  echo "/etc/randomtable is already in place.";
    else
  echo "# /etc/randomtable" > $PKG_INSTALL_ROOT/etc/randomtable
  echo "1121554 # first random number" >> $PKG_INSTALL_ROOT/etc/randomtable
fi

!remove
# randomtable deconstructor
if [ -f $PKG_INSTALL_ROOT/etc/randomtable ]; then
  # the file can be removed if it's unchanged
  if [ egrep "first random number" $PKG_INSTALL_ROOT/etc/
randomtable ]; then
   rm $PKG_INSTALL_ROOT/etc/randomtable;
  fi
fi
```

See Chapter 5 for another example using the `build` class.

The `preserve` *Class Script*

The `preserve` class preserves a package object file by determining whether or not an existing file should be overwritten when the package is installed. Two possible scenarios when using a `preserve` class script are:

- If the file to be installed does not already exist in the target directory, the file will be installed normally.

- If the file to be installed exists in the target directory, a message describing the file exists is displayed, and the file is not installed.

Both scenario outcomes are considered successful by the `preserve` script. A failure occurs only, when in the second scenario, the file is unable to be copied to the target directory.

Starting with the Solaris 7 release, the `i.preserve` script and a copy of this script, `i.CONFIG.prsv`, can be found in the `/usr/sadm/install/scripts` directory with the other class action scripts.

```
#ident "$Header: i.CONFIG.prsv,v 12.1 1997/09/19 $"

error=no
echo "## checking common configuration files"
while read src dest
do
      [ "$src" = /dev/null ] && continue
```

(continued)

```
      if [ -f "$dest" ]
      then
              echo $dest preserved
      else
              echo $dest
              cp $src $dest || error=yes
      fi

done
[ "$error" = yes ] &&
      exit 2
exit 0
```

Figure 3–3   Format of a `preserve` Class Action Script

Modify the script to include the filename or filenames you would like to preserve.

▼ How to Write Class Action Scripts

1. **Make the directory containing your information files the current working directory.**

2. **Assign the package objects in the** `prototype` **file the desired class names. For example, assigning objects to an** `application` **and** `manpage` **class would look like:**

```
f manpage /usr/share/man/manl/myappl.1l
f application /usr/bin/myappl
```

3. **Modify the** `CLASSES` **parameter in the** `pkginfo` **file to contain the class names you want to use in your package. For example, entries for the** `application` **and** `manpage` **classes would look like:**

```
CLASSES=manpage application none
```

Note - The `none` class is always installed first and removed last, regardless of where it appears in the definition of the `CLASSES` parameter.

4. **If you are a creating class action script for a file belonging to the** `sed`**,** `awk`**, or** `build` **class, make the directory containing the package object your current working directory.**

5. **Create the class action scripts or package objects (for files belonging to the** `sed`**,** `awk`**, or** `build` **class). An installation script for a class named** `application` **would be named** `i.application` **and a removal script would be named** `r.application`**.**

   Remember, when a file is part of a class that has a class action script, the script must install the file. The `pkgadd` command does not install files for which a class action script exists, although it does verify the installation. And, if you define a class but do not deliver a class action script, the only action taken for that class is to copy components from the installation medium to the target system (the default `pkgadd` behavior).

6. **Complete** *one* **of the following tasks.**
   - If you have *not* created your `prototype` file, complete the procedure "How to Create a `prototype` File Using the `pkgproto` Command" on page 32, and skip to Step 7 on page 69.
   - If you have already created your `prototype` file, edit it and add an entry for each installation script you just created.

7. **Build your package.**

   See "How to Build a Package" on page 35, if needed.

Where to Go Next

After you build the package, install it to confirm that it installs correctly and verify its integrity. Chapter 4 explains how to do this and provides step-by-step instructions on how to transfer your verified package to a distribution medium.

Verifying and Transferring a Package

This chapter describes how to verify your package's integrity and transfer it to a distribution medium, such as floppy disk or a CD-ROM.

This is a list of the overview information in this chapter:

- "Verifying and Transferring a Package Task Map" on page 71
- "Installing Software Packages" on page 72
- "Verifying the Integrity of a Package" on page 75
- "Displaying Additional Information About Installed Packages" on page 77
- "Removing a Package" on page 83
- "Transferring a Package to a Distribution Medium" on page 84

Verifying and Transferring a Package Task Map

The table below describes the steps you should follow in order to verify your package's integrity and transfer it to a distribution medium.

TABLE 4–1 Verifying and Transferring a Package Task Map

Task	Description	For Instructions, Go To ...
1. Build Your Package	Build your package on disk.	Chapter 2
2. Install Your Package	Test your package by installing it and making sure that it installs without errors.	"How to Install a Package on a Standalone or Server" on page 74
2. Verify Your Package's Integrity	Use the pkgchk command to verify the integrity of your package.	"How to Verify the Integrity of Your Package" on page 75
3. Obtain Other Package Information	*Optional.* Use the `pkginfo` and `pkgparam` commands to perform package-specific verification.	"Displaying Additional Information About Installed Packages" on page 77
4. Remove the Installed Package	Use the `pkgrm` command to remove your installed package from the system.	"How to Remove a Package" on page 83
5. Transfer Your Package to a Distribution Medium	Use the `pkgtrans` command to transfer your package (in package format) to a distribution medium.	"How to Transfer Your Package to a Distribution Medium" on page 84

Installing Software Packages

Software packages are installed using the `pkgadd` command. This command transfers the contents of a software package from the distribution medium or directory and installs it onto a system.

This section provides basic installation instructions for installing your package in order to verify that it installs correctly.

The Installation Software Database

Information for all packages installed on a system is kept in the installation software database. There is an entry for every object in a package, with information such as

the component name, where it resides, and its type. An entry contains a record of the package to which a component belongs; other packages that might reference the component; and information such as path name, where the component resides and the component type. Entries are added and removed automatically by the `pkgadd` and `pkgrm` commands. You can view the information in the database by using the `pkgchk` and the `pkginfo` commands.

Two types of information are associated with each package component. The attribute information describes the component itself. For example, the component's access permissions, owner ID, and group ID are attribute information. The content information describes the contents of the component, such as file size and time of last modification.

The installation software database keeps track of the package status. A package can be either fully installed (it has successfully completed the installation process), or partially installed (it did not successfully complete the installation process).

When a package is partially installed, portions of a package may have been installed before installation was terminated; thus, part of the package is installed, and recorded in the database, and part is not. When you reinstall the package, you are prompted to start at the point where installation stopped because the `pkgadd` command can access the database and detect which portions have already been installed. You can also remove the portions that have been installed, based on the information in the installation software database using the `pkgrm` command.

Interacting with the `pkgadd` Command

If the `pkgadd` command encounters a problem, it first checks the installation administration file for instructions. (See **admin**(4) for more information.) If no instructions exist, or if the relevant parameter in the administration file is set to `ask`, the `pkgadd` displays a message describing the problem and prompts for a reply. The prompt is usually `Do you want to continue with this installation?`. You should respond with `yes`, `no`, or `quit`.

If you have specified more than one package, `no` stops installation of the package being installed but `pkgadd` continues with installation of the other packages. `quit` indicates that `pkgadd` should stop installation of all packages.

Installing Packages on Standalones or Servers in a Homogeneous Environment

This section describes how to install packages on a standalone or a server system in a homogeneous environment.

▼ How to Install a Package on a Standalone or Server

1. **Build your package.**

   See "Building a Package" on page 34, if needed.

2. **Log in to the system as superuser.**

3. **Add the software package to the system.**

   ```
   # pkgadd -d device-name [ pkg-abbrev... ]
   ```

–d *device-name*	Specifies the location of the package. Note that *device-name* can be a full directory path name or the identifiers for a tape, floppy disk, or removable disk.
pkg-abbrev	Is the name of one or more packages (separated by spaces) to be added. If omitted, pkgadd installs all available packages.

Where to Go Next

If you are ready to go to the next task, see "How to Verify the Integrity of Your Package" on page 75.

Example—Installing Packages on Standalones and Servers

To install a software package named pkgA from a tape device named /dev/rmt/0, you would enter the following command:

```
# pkgadd -d /dev/rmt/0 pkgA
```

You can also install multiple packages at the same time, as long as you separate package names with a space, as follows:

```
# pkgadd -d /dev/rmt/0 pkgA pkgB pkgC
```

If you do not name the device on which the package resides, the command checks the default spool directory (/var/spool/pkg). If the package is not there, the installation fails.

Verifying the Integrity of a Package

The `pkgchk` command enables you to check the integrity of packages, whether they are installed on a system or in package format (ready to be installed with the `pkgadd` command). It confirms package structure or the installed files and directories, or displays information about package objects. The `pkgchk` command can list or check the following:

- The package installation scripts.
- The contents or attributes, or both, of objects currently installed on the system.
- The contents of a spooled, uninstalled package.
- The contents or attributes, or both, of objects described in the specified `pkgmap` file.

For more information about this command, refer to **pkgchk**(1M).

The `pkgchk` command performs two kinds of checks. It checks file attributes (the permissions and ownership of a file and major/minor numbers for block or character special devices) and the file contents (the size, checksum, and modification date). By default, the command checks both the file attributes and the file contents.

The `pkgchk` command also compares the file attributes and contents of the installed package against the installation software database. The entries concerning a package may have been changed since the time of installation; for example, another package may have changed a package component. The database reflects that change.

▼ How to Verify the Integrity of Your Package

1. **Install your package.**

   See "How to Install a Package on a Standalone or Server" on page 74, if needed.

2. **Verify the integrity of your package.**

   ```
   # pkgchk  [-v] [-R root-path] [pkg-abbrev...]
   ```

−v	Lists files as they are processed.
−R *root-path*	Specifies the location of the client system's root file system.
pkg-abbrev	Is the name of one or more packages (separated by spaces) to be checked. If omitted, `pkgchk` checks all available packages.

Where to Go Next

If you are ready to go to the next task, see "How to Obtain Information With the `pkginfo` Command" on page 82.

Examples—Verifying the Integrity of a Package

This example shows the command you should use to verify the integrity of an installed package.

```
$ pkgchk pkg-abbrev
$
```

If there are errors, the `pkgchk` command prints them. Otherwise, it does not print anything and returns an exit code of 0. If you do not supply a package abbreviation, then it will check all of the packages on the system.

Alternately, you could use the −v option, which will print a list of files in the package if there are no errors. For example:

```
$ pkgchk -v SUNWcadap
/opt/SUNWcadap
/opt/SUNWcadap/demo
/opt/SUNWcadap/demo/file1
/opt/SUNWcadap/lib
/opt/SUNWcadap/lib/file2
/opt/SUNWcadap/man
/opt/SUNWcadap/man/man1
/opt/SUNWcadap/man/man1/file3.1
/opt/SUNWcadap/man/man1/file4.1
/opt/SUNWcadap/man/windex
/opt/SUNWcadap/srcfiles
/opt/SUNWcadap/srcfiles/file5
/opt/SUNWcadap/srcfiles/file6
$
```

If you need to verify a package that is installed on a client system's root file system, use this command:

```
$ pkgchk -v -R root-path pkg-abbrev
```

Displaying Additional Information About Installed Packages

You can use two other commands to display information about installed packages:

- The `pkgparam` command displays parameter values.

- The `pkginfo` command displays information from the installation software database.

The `pkgparam` Command

The `pkgparam` command enables you to display the values associated with the parameters you specified on the command line. The values are retrieved from either the `pkginfo` file for a specific package, or from the file you name. One parameter value is shown per line. You can display the values only or the parameters and their values.

▼ How to Obtain Information With the `pkgparam` Command

1. **Install your package.**

   See "How to Install a Package on a Standalone or Server" on page 74, if needed.

2. **Display additional information about your package.**

   ```
   # pkgparam [-v] pkg-abbrev [param...]
   ```

−v	Displays the name of the parameter and its value.
pkg-abbrev	Is the name of a specific package.
param	Specifies one or more parameters whose value is displayed.

Where to Go Next

If you are ready to go to the next task, see "How to Remove a Package" on page 83.

Examples—Obtaining Information With the `pkgparam` Command

For example, to display values only, use this command.

```
$ pkgparam SUNWcadap
none
/opt
US/Mountain
/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin
/usr/sadm/sysadm
SUNWcadap
Chip designers need CAD application software to design abc
chips.  Runs only on xyz hardware and is installed in the usr
partition.
system
release 1.0
SPARC
venus990706083849
SUNWcadap
/var/sadm/pkg/SUNWcadap/save
Jul 7 1999 09:58
$
```

To display parameters and their values, use the following command.

```
$ pkgparam -v SUNWcadap
pkgparam -v SUNWcadap
CLASSES='none'
BASEDIR='/opt'
TZ='US/Mountain'
PATH='/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin'
OAMBASE='/usr/sadm/sysadm'
PKG='SUNWcadap'
NAME='Chip designers need CAD application software to design abc chips.
Runs only on xyz hardware and is installed in the usr partition.'
CATEGORY='system'
VERSION='release 1.0'
ARCH='SPARC'
PSTAMP='venus990706083849'
PKGINST='SUNWcadap'
PKGSAV='/var/sadm/pkg/SUNWcadap/save'
INSTDATE='Jul 7 1999 09:58'
$
```

Or, if you want to display the value of a specific parameter, use this format:

```
$ pkgparam SUNWcadap BASEDIR
/opt
$
```

For more information, refer to **pkgparam**(1).

The pkginfo Command

You can display information about installed packages with the pkginfo command. This command has several options that enable you to customize both the format and the contents of the display.

You can request information about any number of package instances.

The Default pkginfo Display

When the pkginfo command is executed without options, it displays the category, package instance, and package name of all packages that have been completely installed on your system. The display is organized by categories as shown in the following example.

```
$ pkginfo
.
.
.
system      SUNWinst      Install Software
system      SUNWipc       Interprocess Communications
system      SUNWisolc     XSH4 conversion for ISO Latin character sets
application SUNWkcspf     KCMS Optional Profiles
application SUNWkcspg     KCMS Programmers Environment
application SUNWkcsrt     KCMS Runtime Environment
.
.
.
$
```

Customizing the Format of the pkginfo Display

You can get a pkginfo display in any of three formats: short, extracted, and long.

The short format is the default. It shows only the category, package abbreviation, and full package name, as shown in "The Default pkginfo Display" on page 79.

The extracted format shows the package abbreviation, package name, package architecture (if available), and package version (if available). Use the –x option to request the extracted format as shown in the next example.

```
$ pkginfo -x
.
.
.
SUNWipc        Interprocess Communications
               (sparc) 11.8.0,REV=1999.08.20.12.37
SUNWisolc      XSH4 conversion for ISO Latin character sets
               (sparc) 1.0,REV=1999.07.10.10.10
SUNWkcspf      KCMS Optional Profiles
               (sparc) 1.1.2,REV=1.5
SUNWkcspg      KCMS Programmers Environment
               (sparc) 1.1.2,REV=1.5
.
.
.
$
```

Using the −l option produces a display in the long format showing all of the available information about a package, as in the following example.

```
$ pkginfo -l SUNWcadap
   PKGINST:  SUNWcadap
      NAME:  Chip designers need CAD application software to
design abc chips.  Runs only on xyz hardware and is installed
in the usr partition.
  CATEGORY:  system
      ARCH:  SPARC
   VERSION:  release 1.0
   BASEDIR:  /opt
    PSTAMP:  system980706083849
  INSTDATE:  Jul 7 1999 09:58
    STATUS:  completely installed
     FILES:     13 installed pathnames
                 6 directories
                 3 executables
             3121 blocks used (approx)
$
```

Parameter Descriptions for the pkginfo Long Format

The table below describes the package parameters that can be displayed for each package. A parameter and its value are displayed only when the parameter has a value assigned to it.

TABLE 4–2 Package Parameters

Parameter	Description
ARCH	The architecture supported by this package.
BASEDIR	The base directory in which the software package resides (shown if the package is relocatable).
CATEGORY	The software category, or categories, of which this package is a member (for example, `system` or `application`).
CLASSES	A list of classes defined for a package. The order of the list determines the order in which the classes are installed. Classes listed first will be installed first (on a media by media basis). This parameter may be modified by the request script.
DESC	Text that describes the package.
EMAIL	The electronic mail address for user inquiries.
HOTLINE	Information on how to receive hotline help about this package.
INTONLY	Indicates that the package should only be installed interactively when set to any non-NULL value.
ISTATES	A list of allowable run states for package installation (for example, `S s 1`).
MAXINST	The maximum number of package instances that should be allowed on a machine at the same time. By default, only one instance of a package is allowed.
NAME	The package name, generally text describing the package abbreviation.
ORDER	A list of classes defining the order in which they should be put on the medium. Used by the `pkgmk` command in creating the package. Classes not defined in this parameter are placed on the medium using the standard ordering procedures.
PKGINST	Abbreviation for the package being installed.
PSTAMP	The production stamp for this package
RSTATES	A list of allowable run states for package removal (for example, `S s 1`).

TABLE 4–2 Package Parameters *(continued)*

ULIMIT	If set, this parameter is passed as an argument to the ulimit command, which establishes the maximum size of a file during installation. This applies only to files created by procedure scripts.
VENDOR	The name of the vendor who supplied the software package.
VERSION	The version of this package.
VSTOCK	The vendor-supplied stock number.

For detailed information about the pkginfo command, refer to the **pkginfo**(1) man page.

▼ How to Obtain Information With the pkginfo Command

1. **Install your package.**

   See "How to Install a Package on a Standalone or Server" on page 74, if needed.

2. **Display additional information about your package.**

   ```
   # pkginfo [-x | -l] [pkg-abbrev]
   ```

−x	Displays package information in extracted format.
−l	Displays package information in long format.
pkg-abbrev	Is the name of a specific package. If omitted, the pkginfo command displays information about all installed packages, in the default format.

Where to Go Next

If you are ready to go to the next task, see "How to Remove a Package" on page 83.

Removing a Package

Because the `pkgrm` command updates information in the software products database, it is important when you remove a package to use the `pkgrm` command—even though you might be tempted to use the `rm` command instead. For example, you could use the `rm` command to remove a binary executable file, but that is not the same as using `pkgrm` to remove the software package that includes that binary executable. Using the `rm` command to remove a package's files will corrupt the software products database. (If you really only want to remove one file, you can use the `removef` command, which will update the software product database correctly. See **removef**(1) for more information.)

▼ How to Remove a Package

1. **Log in to the system as superuser.**

2. **Remove an installed package.**

   ```
   # pkgrm pkg-abbrev ...
   ```

pkg-abbrev	Is the name of one or more packages (separated by spaces). If omitted, `pkgrm` removes all available packages.

3. **Verify that the package has successfully been removed, use the** `pkginfo` **command.**

   ```
   $ pkginfo | egrep pkg-abbrev
   ```

   If *pkg-abbrev* is installed, the `pkginfo` command returns a line of information about it. Otherwise, `pkginfo` returns the system prompt.

Transferring a Package to a Distribution Medium

The `pkgtrans` command moves packages and performs package format translations. You can use the `pkgtrans` command to perform the following translations for an installable package:

- File system format to datastream format
- Datastream format to file system format
- One file system format to another file system format

▼ How to Transfer Your Package to a Distribution Medium

1. **Build your package, creating a directory format package, if you have not already done so.**

   For more information, see "How to Build a Package" on page 35.

2. **Install your package to verify that it installs correctly.**

   See "How to Install a Package on a Standalone or Server" on page 74, if needed.

3. **Verify your package's integrity.**

   See "How to Verify the Integrity of Your Package" on page 75, "How to Obtain Information With the `pkgparam` Command" on page 77, and "How to Obtain Information With the `pkginfo` Command" on page 82, if needed.

4. **Remove the installed package from the system.**

   See "How to Remove a Package" on page 83, if needed.

5. **Transfer the package (in package format) to a distribution medium.**

   To perform a basic translation, execute the following command:

   ```
   $ pkgtrans device1 device2 [ pkg-abbrev... ]
   ```

device1	Is the name of the device where the package currently resides.
device2	Is the name of the device onto which the translated package will be written.
[*pkg-abbrev*]	Is one or more package abbreviations.

If no package names are given, all packages residing in *device1* are translated and written to *device2*.

Note - If more than one instance of a package resides on *device1*, you must use an instance identifier for the package. For a description of a package identifier, see "Defining a Package Instance" on page 15. When an instance of the package being translated already exists on *device2*, the pkgtrans command does not perform the translation. You can use the −o option to tell the pkgtrans command 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 this check does not apply when *device2* supports a datastream format.

Where to Go Next

At this point you have completed the steps necessary to design, build, verify, and transfer your package. If you are interested in looking at some case studies, see Chapter 5. If you are interested in advanced package design ideas, see Chapter 6.

Package Creation Case Studies

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

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

This is a list of the case studies in this chapter:

Soliciting Input From the Administrator

The package in this case study has three types of objects. The administrator may choose which of the three types to install and where to locate the objects on the installation machine.

87

Techniques

This case study demonstrates the following techniques:

- Using parametric path names (variables in object path names) that are used to establish multiple base directories

  For information on parametric path names, see "Parametric Path Names" on page 23.

- Using a `request` script to solicit input from the administrator

  For information on `request` scripts, see "Writing a `request` Script" on page 54.

- Setting conditional values for an installation parameter

Approach

To set up the selective installation in this case study, you must complete the following tasks:

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

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

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

  Normally when you define a class, you should list that class in the `CLASSES` parameter in the `pkginfo` file. Otherwise, no objects in that class are installed. For this case study, the parameter is initially set to null, which means no objects will get installed. The `CLASSES` parameter will be changed by the `request` script, based on the choices of the administrator. This way, the `CLASSES` parameter is set to only those object types that the administrator wants installed.

Note - Usually it is a good idea to set parameters to a default value. If this package had components common to all three object types, you could assign them to the `none` class, and then set the `CLASSES` parameter equal to `none`.

- Insert parametric path names into the `prototype` file.

  The `request` script sets these environment variables to the value that the administrator provides. Then, the `pkgadd` command resolves these environment variables at installation time and knows where to install the package.

  The three environment variables used in this example are set to their default in the `pkginfo` file and serve the following purposes:

  - `$NCMPBIN` defines the location for object executables

- $NCMPMAN defines the location for man pages
- $EMACS defines the location for emacs executables

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

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

  The request script for this package asks the administrator two questions:

  - Should this part of the package be installed?

    When the answer is yes, the appropriate class name is added to the CLASSES parameter. For example, when the administrator chooses to install the man pages associated with this package, the class man is added to the CLASSES parameter.
  - If so, where should this part of the package be placed?

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

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

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

  When looking at the request script for this case study, notice that the questions are generated by the data validation tools ckyorn and ckpath. For more information on these tools, see **ckyorn**(1) and **ckpath**(1).

Case Study Files

The pkginfo File

```
PKG=ncmp
NAME=NCMP Utilities
CATEGORY=application, tools
BASEDIR=/
ARCH=SPARC
VERSION=RELEASE 1.0, Issue 1.0
CLASSES=""
NCMPBIN=/bin
```

(continued)

```
NCMPMAN=/usr/man
EMACS=/usr/emacs
```

The `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
```

The `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"
```

(continued)

```
` || 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
```

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

These example files show the use of parametric paths to establish multiple base directories. However, the preferred method involves use of the `BASEDIR` parameter which is managed and validated by the `pkgadd` command. Whenever multiple base directories are used, take special care to provide for installation of multiple versions and architectures on the same platform.

Creating a File at Installation and Saving It During Removal

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

Techniques

This case study demonstrates the following techniques:

■ Using classes and class action scripts to perform special actions on different sets of objects

For more information, see "Writing Class Action Scripts" on page 61.

■ Using the `space` file to inform the `pkgadd` command that extra space is required to install this package properly

For more information on the space file, see "Reserving Additional Space on a Target System" on page 47.

■ Using the `installf` command to install a file not defined in the `prototype` and `pkgmap` files

Approach

To create a database file at installation and save a copy on removal for this case study, you must complete the following tasks:

■ Define three classes.

The package in this case study requires the following three classes be defined in the `CLASSES` parameter:

- The standard class of `none`, which contains a set of processes belonging in the subdirectory `bin`.
- The `admin` class, which contains an executable file `config` and a directory containing data files.
- The `cfgdata` class, which contains a directory.

■ Make the package collectively relocatable.

Notice in the `prototype` file that none of the path names begins with a slash or an environment variable. This indicates that they are collectively relocatable.

■ Calculate the amount of space the database file requires and create a `space` file to deliver with the package. This file notifies the `pkgadd` command that the package requires extra space and specifies how much.

■ Create a class action script for the `admin` class (`i.admin`).

The sample script initializes a database using the data files belonging to the `admin` class. To perform this task, it does the following:

- 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 the `installf -f` command to finalize installation of `config.data`

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

■ Create a removal class action script for the `cfgdata` class (`r.cfgdata`).

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

Remember that the input to a removal script is a list of path names to remove. Path names always appear in reverse alphabetical order. This removal script copies files to the directory named $PKGSAV. When all the path names have been processed, the script then goes back and removes all directories and files associated with the cfgdata class.

The outcome of this removal script is to copy config.data to $PKGSAV and then remove the config.data file and the data directory.

Case Study Files

The pkginfo File

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

The prototype File

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

The `space` File

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

The `i.admin` Class Action Script

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

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

The `r.cfgdata` Removal Script

```
# the product manager for this package has suggested that
# the configuration data is so valuable that it should be
# backed up to $PKGSAV before it is removed!
while read path
do
# path names appear in reverse lexical order.
    mv $path $PKGSAV || exit 2
    rm -f $path || exit 2
done
```

(continued)

```
exit 0
```

Defining Package Compatibilities and Dependencies

The package in this case study uses optional information files to define package compatibilities and dependencies, and to present a copyright message during installation.

Techniques

This case study demonstrates the following techniques:

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

For more information on these files, see "Creating Information Files" on page 43.

Approach

To meet the requirements in the description, you must:

- Create a `copyright` file.

  A `copyright` file contains the ASCII text of a copyright message. The message shown in the sample file is displayed on the screen during package installation.

- Create a `compver` file.

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

- Create a `depend` file.

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

Case Study Files

The pkginfo File

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

The copyright File

```
Copyright (c) 1999 company_name
All Rights Reserved.
THIS PACKAGE CONTAINS UNPUBLISHED PROPRIETARY SOURCE CODE OF
company_name.
The copyright notice above does not evidence any
actual or intended publication of such source code
```

The compver File

```
Version 3.0
Version 2.3
Version 2.2
Version 2.1
Version 2.1.1
Version 2.1.3
Version 1.7
```

The `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 rfs Remote File Sharing Utilities
Issue 1 Version 1
P rx Remote Execution Utilities
P sgs Software Generation Utilities
Issue 4 Version 1
P shell Shell Programming Utilities
P sys System Header Files
Release 3.1
```

Modifying a File Using Standard Classes and Class Action Scripts

This case study modifies an existing file during package installation using standard classes and class action scripts. It uses one of three modification methods. The other two methods are described in "Modifying a File Using the `sed` Class and a `postinstall` Script" on page 100 and "Modifying a File Using The `build` Class" on page 103. The file modified is `/etc/inittab`.

Techniques

This case study demonstrates how to use installation and removal class action scripts. For more information, see "Writing Class Action Scripts" on page 61.

Approach

To modify `/etc/inittab` during installation, using classes and class action scripts, you must complete the following tasks:

- Create a class.

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

- Create an `inittab` file.

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

- Create an installation class action script (`i.inittab`).

  Remember that class action scripts must produce the same results each time they are executed. The class action script performs the following procedures:

  - Checks 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 the `init q` command when it receives the `ENDOFCLASS` indicator

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

- Create a removal class action script (`r.inittab`).

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

This case study is more complicated than the next one; see "Modifying a File Using the `sed` Class and a `postinstall` Script" on page 100. Instead of providing two files, three are needed and the delivered `/etc/inittab` file is actually just a place holder containing a fragment of the entry to be inserted. This could have been placed into the `i.inittab` file except that the `pkgadd` command must have a file to pass to the `i.inittab` file. Also, the removal procedure must be placed into a separate file (`r.inittab`). While this method works fine, it is best reserved for cases involving very complicated installations of multiple files. See "Modifying `crontab` Files During Installation" on page 105.

The `sed` program used in "Modifying a File Using the `sed` Class and a `postinstall` Script" on page 100 supports multiple package instances since the comment at the end of the `inittab` entry is based on package instance. The case study in "Modifying a File Using The `build` Class" on page 103 shows a more streamlined approach to editing `/etc/inittab` during installation.

Case Study Files

The `pkginfo` File

```
PKG=case5
NAME=Case Study #5
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1d05
CLASSES=inittab
```

The `prototype` File

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

The `i.inittab` Installation Class Action Script

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

The `r.inittab` Removal Class Action Script

```
# 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
```

The `inittab` File

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

Modifying a File Using the `sed` Class and a `postinstall` Script

This case 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 described in "Modifying a File Using Standard Classes and Class Action Scripts" on page 97 and "Modifying a File Using The `build` Class" on page 103. The file modified is `/etc/inittab`.

Techniques

This case study demonstrates the following techniques:

- Using the `sed` class

  For more information on the `sed` class, see "The `sed` Class Script " on page 65.

- Using a `postinstall` script

  For more information on this script, see "Writing Procedure Scripts" on page 59.

Approach

To modify `/etc/inittab` at the time of installation, using the `sed` class, you must complete the following tasks:

- 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 the `sed` script is named `/etc/inittab`. There are no requirements for the mode, owner, and group of a `sed` script (represented in the sample `prototype` by question marks). The file type of the `sed` script must be `e` (indicating that it is editable).

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

  As shown in the example file, `sed` is the only class being installed. However, it could be one of any number of classes.

- Create a `sed` class action script.

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

- Create a `postinstall` script.

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

This approach to editing `/etc/inittab` during installation has one drawback; you have to deliver a full script (the `postinstall` script) simply to perform the `init q` command.

Case Study Files

The `pkginfo` File

```
PKG=case4
NAME=Case Study #4
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1d05
```

(continued)

```
CLASSES=sed
```

The `prototype` File

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

The `sed` Class Action Script (`/etc/inittab`)

```
!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
```

The `postinstall` Script

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

Modifying a File Using The `build` Class

This case 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 described in "Modifying a File Using Standard Classes and Class Action Scripts" on page 97 and "Modifying a File Using the `sed` Class and a `postinstall` Script" on page 100. The file modified is `/etc/inittab`.

Techniques

This case study demonstrates how to use the `build` class. For more information on the `build` class, see "The `build` Class Script " on page 66.

Approach

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

The `build` class script 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` class script that installation actions are defined by testing for this argument.

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

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

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

- Create the `build` class script.

The sample `build` class script performs the following procedures:

- Edits the `/etc/inittab` file to remove any existing changes for this package. Notice that the file name `/etc/inittab` is hardcoded 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 describe where that entry came from.
- Executes the `init q` command.

This solution addresses the drawbacks described in the case studies in "Modifying a File Using Standard Classes and Class Action Scripts" on page 97 and "Modifying a File Using the `sed` Class and a `postinstall` Script" on page 100. Only one short file is needed (beyond the `pkginfo` and `prototype` files). The file works with multiple instances of a package since the PKGINST parameter is used, and no `postinstall` script is required since the `init q` command can be executed from the `build` class script.

Case Study Files

The `pkginfo` File

```
PKG=case6
NAME=Case Study #6
CATEGORY=applications
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1d05
CLASSES=build
```

The `prototype` File

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

The Build File

```
# 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
```

Modifying `crontab` Files During Installation

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

Techniques

This case study demonstrates the following techniques:

- Using classes and class action scripts

  For more information, see "Writing Class Action Scripts" on page 61.

- Using the `crontab` command within a class action script

Approach

The most efficient way to edit more than one file during installation is to define a class and provide a class action script. If you used the `build` class approach, you would need to deliver one `build` class script for each `crontab` file edited. Defining a `cron` class provides a more general approach. To edit `crontab` files with this approach, you must:

- Define the `crontab` files that are to be edited in the `prototype` file.

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

- Create the `crontab` files for the package.

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

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

  The sample `i.cron` script performs the following procedures:

  - Determines the user ID (UID).

    The `i.cron` script sets the variable *user* to the base name of the `cron` class script being processed. That name is the UID. For example, the base name of `/var/spool/cron/crontabs/root` is root, which is also the UID.

  - Executes `crontab` using the UID and the −l option.

    Using the −l option tells `crontab` to send the contents of the `crontab` file for the defined user to the standard output.

  - Pipes the output of the `crontab` command to a `sed` script that removes any previous entries added with this installation technique.

  - Puts the edited output into a temporary file.

  - Adds the data file for the root UID (that was delivered with the package) to the temporary file and adds a tag so you will know where these entries came from.

  - Executes `crontab` with the same UID and gives it the temporary file as input.

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

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

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

Case Study Files

The `i.cron` and `r.cron` scripts described below are executed by superuser. Editing another user's `crontab` file as superuser may have unpredictable results. If necessary, change the following entry in each script:

```
crontab $user < /tmp/$$crontab ||
```

to

```
su $user -c "crontab /tmp/$$crontab" ||
```

The `pkginfo` Command

```
PKG=case7
NAME=Case Study #7
CATEGORY=application
BASEDIR=/opt
ARCH=SPARC
VERSION=Version 1.0
CLASSES=cron
```

The `prototype` File

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

The `i.cron` Installation Class Action Script

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

The `r.cron` Removal Class Action Script

```
# 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
```

`crontab` File #1

```
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
```

`crontab` File #2

```
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
```

Note - If editing of a group of files will increase total file size by more than 10K,
supply a `space` file so the `pkgadd` command can allow for this increase. For more
information on the `space` file, see "Reserving Additional Space on a Target System"
on page 47.

Installing and Removing a Driver With Procedure Scripts

This package installs a driver.

Techniques

This case study demonstrates the following techniques:

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

For more information on these scripts, see "Writing Procedure Scripts" on page 59.

Approach

- Create a `request` script.

  The `request` script determines where the administrator wants the driver objects to be installed, by questioning the administrator and assigning the answer to the `$KERNDIR` parameter.

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

- Create a `postinstall` script.

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

  - Uses the `add_drv` command to load the driver into the system.
  - Creates a link for the device using the `installf` command.
  - Finalizes the installation using the `installf -f` command.
  - Creates a `preremove` script.

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

Case Study Files

The `pkginfo` File

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

The `prototype` File

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

In this example, the executable module for the driver is named `buffer`; the `add_drv` command operates on this file. The kernel uses the configuration file, `buffer.conf`, to help configure the driver.

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

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

- Since no special treatment is required for the package objects, you can put them into the standard `none` class. The CLASSES parameter is set to `none` in the `pkginfo` file.

- The path names for `buffer` and `buffer.conf` begin with the variable $KERNDIR. This variable is set in the `request` script and allows the administrator to decide where the driver files should be installed. The default directory is `/kernel/drv`.

- There is an entry for the `postinstall` script (the script that will perform the driver installation).

The `request` Script

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

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

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

The `postinstall` Script

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

The `preremove` Script

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

Installing a Driver Using the `sed` Class and Procedure Scripts

This case study describes how to install a driver using the `sed` class and procedure scripts. It is also different from the previous case study (see "Installing and Removing a Driver With Procedure Scripts" on page 109) because this package is made up of both absolute and relocatable objects.

Techniques

This case study demonstrates the following techniques:

- Building a `prototype` file with both absolute and relocatable objects.

  For more information on building a `prototype` file, see "Creating a `prototype` File" on page 20.

- Using a `postinstall` script

  For more information on this script, see "Writing Procedure Scripts" on page 59.

- Using a `preremove` script

  For more information on this script, see "Writing Procedure Scripts" on page 59.

- Using a `copyright` file

  For more information on this file, see "Writing a Copyright Message " on page 46.

Approach

- Create a `prototype` file containing both absolute and relocatable package objects.

  This is discussed in detail in "The `prototype` File" on page 113.

- 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/devlink.tab` and so the `sed` script is named `/etc/devlink.tab`. There are no requirements for the mode, owner, and group of a `sed` script (represented in the sample `prototype` by question marks). The file type of the `sed` script must be `e` (indicating that it is editable).

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

- Create a `sed` class action script (`/etc/devlink.tab`).

- Create a `postinstall` script.

  The `postinstall` script needs to execute the `add_drv` command to add the device driver to the system.

- Create a `preremove` script.

  The `preremove` script needs to execute the `rem_drv` command to remove the device driver from the system, prior to the package being removed.

- Create a `copyright` file.

  A `copyright` file contains the ASCII text of a copyright message. The message shown in the sample file is displayed on the screen during package installation.

Case Study Files

The `pkginfo` File

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

The `prototype` File

For example, this case study uses the hierarchical layout of the package objects shown in the figure below.

```
                                    pkg
          ┌──────────────┬──────────────────────────┬──────────────┐
      pkginfo           etc                        usr           SUNWsst
      postinstall        │                          │               │
      preremove     devlink.tab          ┌──────────┴─────────┐
      copyright                        kernel              include    sstest.c
                                         │                     │
                                        drv                   sys
                                         │                     │
                                        sst                  scsi
                                     sst.conf                  │
                                                            targets
                                                               │
                                                           sst_def.h
```
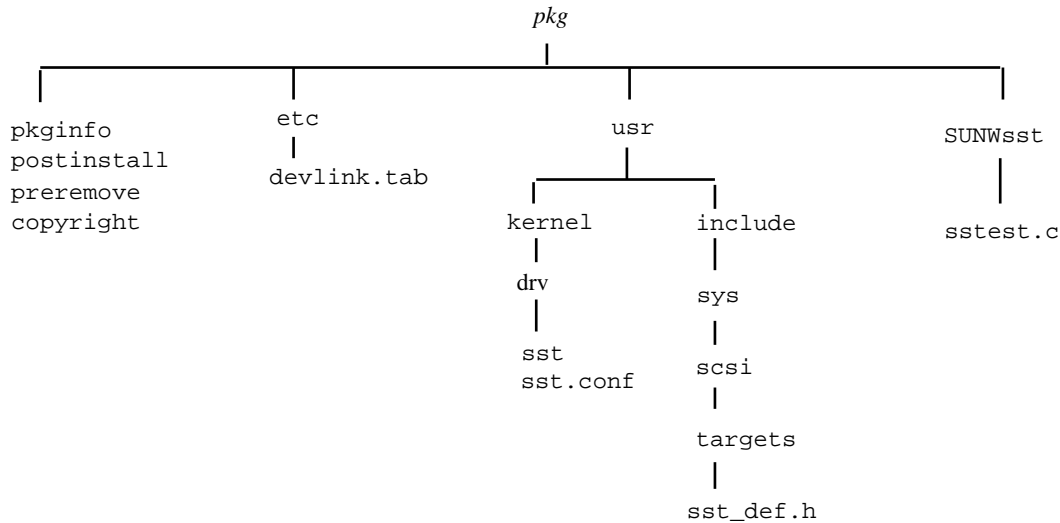
Figure 5–1   Hierarchical Package Directory Structure

The package objects are installed in the same places as they are in the pkg directory above. The driver modules (sst and sst.conf) are installed into /usr/kernel/drv and the include file is installed into /usr/include/sys/scsi/targets. The sst, sst.conf, and sst_def.h files are absolute objects. The test program, sstest.c, and its directory SUNWsst are relocatable; their installation location is set by the BASEDIR parameter.

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

From the pkg directory, run the pkgproto command as follows:

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

The output from the above command looks like this:

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

(continued)

```
d none SUNWsst 0775 pms mts
f none SUNWsst/sstest.c 0664 pms mts
```

This `prototype` file is not yet complete. To complete this file, you need to make the following modifications:

- Insert the entries for the control files (file type `i`), because they have a different format than the other package objects.

- Remove entries for directories that already exist on the target system.

- Change the access permission and ownership for each entry.

- Prepend a slash to the absolute package objects.

This is the final `prototype` file:

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

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

The `sed` Class Action Script (`/etc/devlink.tab`)

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

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

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

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

The postinstall Installation Script

In this example, all the script needs to do is run the add_drv command.

```
# Postinstallation script for SUNWsst
# This does not apply to a client.
if [$PKG_INSTALL_ROOT = "/" -o -z $PKG_INSTALL_ROOT]; then
    SAVEBASE=$BASEDIR
    BASEDIR=''''; export BASEDIR
    /usr/sbin/add_drv sst
    STATUS=$?
    BASEDIR=$SAVEBASE; export BASEDIR
    if [ $STATUS -eq 0 ]
    then
       exit 20
    else
       exit 2
    fi
else
    echo "This cannot be installed onto a client."
    exit 2
fi
```

The add_drv command uses the BASEDIR parameter, so the script has to unset BASEDIR before running the command, and restore it afterwards.

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

The exit code from the postinstall script is significant. The exit code 20 tells the pkgadd command to tell the user to reboot the system (necessary after installing a

driver), and the exit code `2` tells the `pkgadd` command to tell the user that the installation partially failed.

The `preremove` Removal Script

In the case of this driver example, it removes the links in `/dev` and runs the `rem_drv` command on the driver.

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

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

exit
```

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

The `copyright` File

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

```
 Copyright (c) 1999 Drivers-R-Us, Inc.
 10 Device Drive, Thebus, IO 80586

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

Advanced Package Creation Techniques

The full capabilities of System V packaging as implemented in the Solaris operating environment provide a powerful tool for the installation of software products. As a package designer, you can take advantage of these capabilities. Packages that are not part of the Solaris operating environment (unbundled packages) may use the class mechanism to customize server/client installations. Relocatable packages can be designed to accommodate the desires of the administrator. A complex product can be delivered as a set of composite packages that automatically resolve package dependencies. Upgrading and patching may be customized by the package designer. Patched packages can be delivered in the same way as unpatched packages, and the backout archives can also be included in the product.

This is a list of the overview information in this chapter.

Specifying the Base Directory

You can use several methods to specify where a package will be installed, and it is important to be able to change the installation base dynamically at install time. If this is accomplished correctly, an administrator can install multiple versions and multiple architectures without complications.

This section discusses common methods first, followed by approaches that enhance installations to heterogeneous systems.

The Administrative Defaults File

Administrators responsible for installing packages can use administration files to control package installation. However, as a package designer, you need to know about administration files and how an administrator can alter your intended package installation.

An administration file tells the `pkgadd` command whether to perform any of the checks or prompts that it normally does. Consequently, administrators should fully understand a package's installation process and the scripts involved before using administration files.

A basic administrative defaults file is shipped with the SunOS operating system in `/var/sadm/install/admin/default`. This is the file that establishes the most basic level of administrative policy as regards the installation of software products. The file looks like this as shipped:

```
#ident "@(#)default
1.4 92/12/23 SMI" /* SVr4.0 1.5.2.1 */
mail=
instance=unique
partial=ask
runlevel=ask
idepend=ask
rdepend=ask
space=ask
setuid=ask
conflict=ask
action=ask
basedir=default
```

The administrator may edit this file to establish new default behaviors, or create a different administration file and specify its existence by using the −a option to the `pkgadd` command.

Eleven parameters can be defined in an administration file, but not all need to be defined. For more information, see **admin**(4).

The `basedir` parameter specifies how the base directory will be derived when a package is installed. Most administrators leave this as `default`, but `basedir` can be set to one of the following:

- `ask`, which means always ask the administrator for a base directory
- An absolute path name

- An absolute path name containing the $PKGINST construction, which means always install to a base directory derived from the package instance

Note - If the pkgadd command is called with the argument –a none, it always asks the administrator for a base directory. Unfortunately, this also sets *all* parameters in the file to the default value of quit, which can result in additional problems.

Becoming Comfortable With Uncertainty

An administrator has control over all packages being installed on a system by using an administration file. Unfortunately, an alternate administrative defaults file is often provided by the *package designer*, bypassing the wishes of the administrator.

Package designers sometimes include an alternate administration file so that they, not the administrator, control a package's installation. Because the basedir entry in the administrative defaults file overrides all other base directories, it provides a simple method for selecting the appropriate base directory at install time. In all versions of the Solaris operating environment prior to the Solaris 2.5 release, this was considered the simplest method for controlling the base directory.

However, it is necessary for you to accept the administrator's desires regarding the installation of the product. Providing a temporary administrative defaults file for the purpose of controlling the installation leads to mistrust on the part of administrators. You should use a request script and checkinstall script to control these installations under the supervision of the administrator. If the request script faithfully involves the administrator in the process, System V packaging will serve both administrators and package designers.

Using the BASEDIR Parameter

The pkginfo file for any relocatable package must include a default base directory in the form of an entry like this:

```
BASEDIR=absolute_path
```

This is only the default base directory; it can be changed by the administrator during installation.

While some packages require more than one base directory, the advantage to using this parameter to position the package is because the base directory is guaranteed to be in place and writable as a valid directory by the time installation begins. The correct path to the base directory for the server and client is available to all procedure scripts in the form of reserved environment variables, and the pkginfo -r SUNWstuf command displays the current installation base for the package.

In the `checkinstall` script, BASEDIR is the parameter exactly as defined in the `pkginfo` file (it has not been conditioned yet). In order to inspect the target base directory, the `${PKG_INSTALL_ROOT}$BASEDIR` construction is required. This means that the `request` or `checkinstall` script can change the value of BASEDIR in the installation environment with predictable results. By the time the `preinstall` script is called, the BASEDIR parameter is the fully conditioned pointer to the actual base directory on the target system, even if the system is a client.

Note - The `request` script utilizes the BASEDIR parameter differently for different releases of the SunOS operating system. In order to test a BASEDIR parameter in a `request` script, the following code should be used to determine the actual base directory in use.

```
# request script
constructs base directory
if [ ${CLIENT_BASEDIR} ]; then
    LOCAL_BASE=$BASEDIR
else
    LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR
fi
```

Using Parametric Base Directories

If a package requires multiple base directories, you can establish them with parametric path names. This method has become quite popular, although it has the following drawbacks.

- A package with parametric path names usually behaves like an absolute package but is treated by the `pkgadd` command like a relocatable package. The BASEDIR parameter must be defined even if it is not used.

- The administrator cannot ascertain the installation base for the package using the System V utilities (the `pkginfo -r` command will not work).

- The administrator cannot use the established method to relocate the package (it is called relocatable but it acts absolute).

- Multiple architecture or multiple version installations require contingency planning for each of the target base directories which often means multiple complex class action scripts.

While the parameters that determine the base directories are defined in the `pkginfo` file, they may be modified by the `request` script. That is one of the primary reasons for the popularity of this approach. The drawbacks, however are chronic and you should consider this configuration a last resort.

Examples—Using Parametric Base Directories

The pkginfo *File*

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/
EZDIR=/usr/stuf/EZstuf
HRDDIR=/opt/SUNWstuf/HRDstuf
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none
PSTAMP=hubert980707141632
```

The pkgmap *File*

```
: 1 1758
1 d none $EZDIR 0775 root bin
1 f none $EZDIR/dirdel 0555 bin bin 40 773 751310229
1 f none $EZDIR/usrdel 0555 bin bin 40 773 751310229
1 f none $EZDIR/filedel 0555 bin bin 40 773 751310229
1 d none $HRDDIR 0775 root bin
1 f none $HRDDIR/mksmart 0555 bin bin 40 773 751310229
1 f none $HRDDIR/mktall 0555 bin bin 40 773 751310229
1 f none $HRDDIR/mkcute 0555 bin bin 40 773 751310229
1 f none $HRDDIR/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc ? ? ?
1 d none /etc/rc2.d ? ? ?
1 f none /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

Managing the Base Directory

Any package that is available in multiple versions or for multiple architectures
should be designed to *walk* the base directory, if needed. Walking a base directory

means that if a previous version or a different architecture of the package being installed already exists in the base directory, the package being installed resolves this issue, perhaps by creating a new base directory with a slightly different name. The `request` and `checkinstall` scripts in the Solaris 2.5 and compatible releases have the ability to modify the `BASEDIR` environment variable. This is not true for any prior version of the Solaris operating environment.

Even in older versions of the Solaris operating environment, the `request` script had the authority to redefine directories within the installation base. The `request` script can do this in a way that still supports most administrative preferences.

Accommodating Relocation

While you can select base directories for various packages that are guaranteed unique to an architecture and version, this leads to unnecessary levels of directory hierarchy. For example, for a product designed for SPARC and IA based processors, you could organize the base directories by processor and version as shown below.

Base Directory	Version and Processor
/opt/SUNWstuf/sparc/1.0	Version 1.0, SPARC
/opt/SUNWstuf/sparc/1.2	Version 1.2, SPARC
/opt/SUNWstuf/IA/1.0	Version 1.0, IA

This is okay and it does work, but you are treating names and numbers as though they mean something to the administrator. A better approach is to do this automatically *after* explaining it to the administrator and obtaining permission.

This means that you can do the whole job in the package without requiring the administrator to do it manually. You can assign the base directory arbitrarily and then transparently establish the appropriate client links in a `postinstall` script. You can also use the `pkgadd` command to install all or part of the package to the clients in the `postinstall` script. You can even ask the administrator which users or clients need to know about this package and automatically update `PATH` environment variables and `/etc` files. This is completely acceptable as long as whatever the package does upon installation, it undoes upon removal.

Walking Base Directories

You can take advantage of two methods for controlling the base directory at install time. The first is best for new packages that will install only to Solaris 2.5 and compatible releases; it provides very useful data for the administrator and supports multiple installed versions and architectures and requires minimal special work. The second method can be used by any package and makes use of the `request` script's inherent control over build parameters to ensure successful installations.

Using the `BASEDIR` Parameter

The `checkinstall` script can select the appropriate base directory at install time, which means that the base directory can be placed very low in the directory tree. This example increments the base directory sequentially, leading to directories of the form `/opt/SUNWstuf`, `/opt/SUNWstuf.1`, and `/opt/SUNWstuf.2`. The administrator can use the `pkginfo` command to determine which architecture and version are installed in each base directory.

If the `SUNWstuf` package (containing a set of utilities that do stuff) uses this method, its `pkginfo` and `pkgmap` files would look like this.

The `pkginfo` *File*

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt/SUNWstuf
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none daemon
PSTAMP=hubert990707141632
```

The `pkgmap` *File*

```
: 1 1758
1 d none EZstuf 0775 root bin
1 f none EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none HRDstuf 0775 root bin
```

```
1 f none HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc ? ? ?
1 d none /etc/rc2.d ? ? ?
1 f daemon /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
1 i i.daemon 509 39560 752978103
1 i r.daemon 320 24573 742152591
```

Example—Analysis Scripts That Walk a BASEDIR

Assume that the IA version of SUNWstuf is already installed on the server in /opt/SUNWstuf. When the administrator uses the pkgadd command to install the SPARC version, the request script needs to detect the existence of the IA version and interact with the administrator regarding the installation.

Note - The base directory could be walked without administrator interaction in a checkinstall script, but if arbitrary operations like this happen too often, administrators lose confidence in the process.

The request script and checkinstall script for a package that handle this situation might look like this.

The request Script

```
# request script
for SUNWstuf to walk the BASEDIR parameter.

PATH=/usr/sadm/bin:${PATH} # use admin utilities

GENMSG="The base directory $LOCAL_BASE already contains a \
different architecture or version of $PKG."

OLDMSG="If the option \"-a none\" was used, press the  \
key and enter an unused base directory when it is requested."
```

(continued)

```
OLDPROMPT="Do you want to overwrite this version? "

OLDHELP="\"y\" will replace the installed package, \"n\" will \
stop the installation."

SUSPEND="Suspending installation at user request using error \
code 1."

MSG="This package could be installed at the unused base directory $WRKNG_BASE."

PROMPT="Do you want to use to the proposed base directory? "

HELP="A response of \"y\" will install to the proposed directory and continue, \
\"n\" will request a different directory. If the option \"-a none\" was used, \
press the  key and enter an unused base directory when it is requested."

DIRPROMPT="Select a preferred base directory ($WRKNG_BASE) "

DIRHELP="The package $PKG will be installed at the location entered."

NUBD_MSG="The base directory has changed. Be sure to update \
any applicable search paths with the actual location of the \
binaries which are at $WRKNG_BASE/EZstuf and $WRKNG_BASE/HRDstuf."

OldSolaris=""
Changed=""
Suffix="0"

#
# Determine if this product is actually installed in the working
# base directory.
#
Product_is_present () {
   if [ -d $WRKNG_BASE/EZstuf -o -d $WRKNG_BASE/HRDstuf ]; then
      return 1
   else
      return 0
   fi
}

if [ ${BASEDIR} ]; then
   # This may be an old version of Solaris. In the latest Solaris
   # CLIENT_BASEDIR won't be defined yet. In older version it is.
   if [ ${CLIENT_BASEDIR} ]; then
      LOCAL_BASE=$BASEDIR
      OldSolaris="true"
   else # The base directory hasn't been processed yet
      LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR
fi

WRKNG_BASE=$LOCAL_BASE

 # See if the base directory is already in place and walk it if
 # possible
```

(continued)

Advanced Package Creation Techniques **127**

```
while [ -d ${WRKNG_BASE} -a Product_is_present ]; do
    # There is a conflict
    # Is this an update of the same arch & version?
    if [ ${UPDATE} ]; then
        exit 0 # It's out of our hands.
    else
        # So this is a different architecture or
        # version than what is already there.
        # Walk the base directory
        Suffix=`expr $Suffix + 1`
        WRKNG_BASE=$LOCAL_BASE.$Suffix
        Changed="true"
    fi
done

 # So now we can propose a base directory that isn't claimed by
 # any of our other versions.
if [ $Changed ]; then
    puttext "$GENMSG"
    if [ $OldSolaris ]; then
        puttext "$OLDMSG"
        result=`ckyorn -Q -d "a" -h "$OLDHELP" -p "$OLDPROMPT"`
        if [ $result="n" ]; then
            puttext "$SUSPEND"
            exit 1 # suspend installation
        else
            exit 0
        fi
    else # The latest functionality is available
        puttext "$MSG"
        result=`ckyorn -Q -d "a" -h "$HELP" -p "$PROMPT"`
        if [ $? -eq 3]; then
            echo quitinstall >> $1
            exit 0
        fi

        if [ $result="n" ]; then
            WRKNG_BASE=`ckpath -ayw -d "$WRKNG_BASE" \
            -h "$DIRHELP" -p "$DIRPROMPT"`
        else if [ $result="a" ]
            exit 0
        fi
        fi
        echo "BASEDIR=$WRKNG_BASE" >> $1
        puttext "$NUBD_MSG"
    fi
fi
exit 0
```

```
# checkinstall
script for SUNWstuf to politely suspend

grep quitinstall $1
if [ $? -eq 0 ]; then
 exit 3  # politely suspend installation
fi

exit 0
```

This approach would not work very well if the base directory was simply `/opt`. This package has to call out the `BASEDIR` more precisely since `/opt` would be difficult to walk. In fact, depending on the mount scheme, it may not be possible. The example walks the base directory by creating a new directory under `/opt`, which does not introduce any problems.

This example uses a `request` script and a `checkinstall` script even though versions of Solaris prior to the 2.5 release cannot run a `checkinstall` script. The `checkinstall` script in this example is used for the purpose of politely halting the installation in response to a private message in the form of the string "quitinstall." If this script executes under the Solaris 2.3 release, the `checkinstall` script is ignored and the `request` script halts the installation with an error message.

Remember that prior to the Solaris 2.5 and compatible releases, the `BASEDIR` parameter is a read-only parameter and cannot be changed by the `request` script. For this reason, if an old version of the SunOS operating system is detected (by testing for a conditioned `CLIENT_BASEDIR` environment variable), the `request` script has only two options—continue or quit.

Using Relative Parametric Paths

If your software product might be installed on older versions of the SunOS operating system, the `request` script needs to do all the necessary work. This approach can also be used to manipulate multiple directories. If additional directories are required, they still need to be included under a single base directory in order to provide an easily administrable product. While the `BASEDIR` parameter does not provide the level of granularity available in the latest Solaris release, your package can still walk the base directory by using the `request` script to manipulate parametric paths. This is how the `pkginfo` and `pkgmap` files might look.

The `pkginfo` *File*

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt
SUBBASE=SUNWstuf
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none daemon
PSTAMP=hubert990707141632
```

The `pkgmap` *File*

```
: 1 1758
1 d none $SUBBASE/EZstuf 0775 root bin
1 f none $SUBBASE/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none $SUBBASE/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none $SUBBASE/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none $SUBBASE/HRDstuf 0775 root bin
1 f none $SUBBASE/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none $SUBBASE/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none $SUBBASE/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none $SUBBASE/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc ? ? ?
1 d none /etc/rc2.d ? ? ?
1 f daemon /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
1 i i.daemon 509 39560 752978103
1 i r.daemon 320 24573 742152591
```

This example is not perfect. A `pkginfo -r` command returns `/opt` for the
installation base, which is pretty ambiguous. Many packages are in `/opt`, but at least
it is a meaningful directory. Just like the previous example, this next example fully
supports multiple architectures and versions. The `request` script can be tailored to
the needs of the specific package and resolve whatever dependencies are applicable.

Example—A `request` Script That Walks a Relative Parametric Path

```
# request script
for SUNWstuf to walk a parametric path

PATH=/usr/sadm/bin:${PATH} # use admin utilities

MSG="The target directory $LOCAL_BASE already contains \
different architecture or version of $PKG. This package \
could be installed at the unused target directory $WRKNG_BASE."

PROMPT="Do you want to use to the proposed directory? "

HELP="A response of \"y\" will install to the proposed directory \
and continue, \"n\" will request a different directory. If \
the option \"-a none\" was used, press the <RETURN> key and \
enter an unused base directory when it is requested."

DIRPROMPT="Select a relative target directory under $BASEDIR/"

DIRHELP="The package $PKG will be installed at the location entered."

SUSPEND="Suspending installation at user request using error \
code 1."

NUBD_MSG="The location of this package is not the default. Be \
sure to update any applicable search paths with the actual \
location of the binaries which are at $WRKNG_BASE/EZstuf \
and $WRKNG_BASE/HRDstuf."

Changed=""
Suffix="0"

#
# Determine if this product is actually installed in the working
# base directory.
#
Product_is_present () {
    if [ -d $WRKNG_BASE/EZstuf -o -d $WRKNG_BASE/HRDstuf ]; then
        return 1
    else
        return 0

    fi
}

if [ ${BASEDIR} ]; then
    # This may be an old version of Solaris. In the latest Solaris
    # CLIENT_BASEDIR won't be defined yet. In older versions it is.
    if [ ${CLIENT_BASEDIR} ]; then
        LOCAL_BASE=$BASEDIR/$SUBBASE
    else # The base directory hasn't been processed yet
        LOCAL_BASE=${PKG_INSTALL_ROOT}$BASEDIR/$SUBBASE
    fi
```

(continued)

```
WRKNG_BASE=$LOCAL_BASE

# See if the base directory is already in place and walk it if
# possible
while [ -d ${WRKNG_BASE} -a Product_is_present ]; do
   # There is a conflict
   # Is this an update of the same arch & version?
   if [ ${UPDATE} ]; then
      exit 0 # It's out of our hands.
   else
      # So this is a different architecture or
      # version than what is already there.
      # Walk the base directory
      Suffix=`expr $Suffix + 1`
      WRKNG_BASE=$LOCAL_BASE.$Suffix
      Changed="true"
   fi
done

# So now we can propose a base directory that isn't claimed by
# any of our other versions.
if [ $Changed ]; then
   puttext "$MSG"
   result=`ckyorn -Q -d "a" -h "$HELP" -p "$PROMPT"`
   if [ $? -eq 3 ]; then
      puttext "$SUSPEND"
      exit 1
   fi

   if [ $result="n" ]; then
      WRKNG_BASE=`ckpath -lyw -d "$WRKNG_BASE" -h "$DIRHELP" \
      -p "$DIRPROMPT"`

      elif [ $result="a" ]; then
         exit 0
      else
         exit 1
      fi
      echo SUBBASE=$SUBBASE.$Suffix >> $1
      puttext "$NUBD_MSG"
   fi
fi
exit 0
```

Supporting Relocation in a Heterogeneous Environment

The original concept behind System V packaging assumed one architecture per system. The concept of a server did not play a role in the design. Now, of course, a single server may provide support for several architectures, which means there may be several copies of the same software on a server, each for a different architecture. While Solaris packages are sequestered within recommended file system boundaries (for example, `/` and `/usr`), with product databases on the server as well as each client, not all installations necessarily support this division. Certain implementations support an entirely different structure and imply a common product database. While pointing the clients to different versions is straightforward, actually installing System V packages to different base directories can introduce complications for the administrator.

When you design your package, you should also consider the common methods administrators use for introducing new software versions. Administrators often seek to install and test the latest version side-by-side with the currently installed version. The procedure involves installing the new version to a different base directory than the current version and directing a handful of non-critical clients to the new version as a test. As confidence builds, the administrator redirects more and more clients to the new version. Eventually, the administrator retains the old version only for emergencies and then finally deletes it.

What this means is that packages destined for modern heterogeneous systems must support true relocation in the sense that the administrator may put them any reasonable place on the file system and still see full functionality. The Solaris 2.5 and compatible releases provide a number of useful tools which allow multiple architectures and versions to install cleanly to the same system. Solaris 2.4 and compatible versions also support true relocation but accomplishing the task is not quite as obvious.

Traditional Approach

Relocatable Packages

The System V ABI implies that the original intention behind the relocatable package was to make installing the package more convenient for the administrator. Now the need for relocatable packages goes much further. Convenience is not the only issue, rather it is quite possible that during the installation an active software product is already installed in the default directory. A package that is not designed to deal with this situation either overwrites the existing product or fails to install. However, a

package designed handle multiple architectures and multiple versions can install
smoothly and offer the administrator a wide range of options that are fully
compatible with existing administrative traditions.

In some ways the problem of multiple architectures and the problem of multiple
versions is the same. It must be possible to install a variant of the existing package
side by side with other variants, and direct clients or standalone consumers of
exported file systems to any one of those variants, without degraded functionality.
While Sun has established methods for dealing with multiple architectures on a
server, the administrator may not adhere to those recommendations. All packages
need to be capable of complying with the administrators' reasonable wishes
regarding installation.

Example-A Traditional Relocatable Package

This example shows what a traditional relocatable package may look like. The
package is to be located in `/opt/SUNWstuf`, and its `pkginfo` file and `pkgmap` file
might look like this.

The `pkginfo` *File*

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none
PSTAMP=hubert990707141632
```

The `pkgmap` *File*

```
: 1 1758
1 d none SUNWstuf 0775 root bin
1 d none SUNWstuf/EZstuf 0775 root bin
1 f none SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
```

(continued)

```
1 d none SUNWstuf/HRDstuf 0775 root bin
1 f none SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

This is referred to as the traditional method because every package object is installed to the base directory defined by the BASEDIR parameter from the pkginfo file. For example, the first object in the pkgmap file is installed as the directory /opt/SUNWstuf.

Absolute Packages

An absolute package is one that installs to a particular root (/) file system. These packages are difficult to deal with from the standpoint of multiple versions and architectures. As a general rule, all packages should be relocatable. There are, however very good reasons to include absolute elements in a relocatable package.

Example-A Traditional Absolute Package

If the SUNWstuf package was an absolute package, the BASEDIR parameter should not be defined in the pkginfo file, and the pkgmap file would look like this.

The pkgmap *File*

```
: 1 1758
1 d none /opt ? ? ?
1 d none /opt/SUNWstuf 0775 root bin
1 d none /opt/SUNWstuf/EZstuf 0775 root bin
1 f none /opt/SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none /opt/SUNWstuf/HRDstuf 0775 root bin
1 f none /opt/SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
```

(continued)

```
1 f none /opt/SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none /opt/SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

In this example, if the administrator specified an alternate base directory during installation, it would be ignored by the pkgadd command. This package always installs to /opt/SUNWstuf of the target system.

The –R argument to the pkgadd command works as expected. For example,

```
pkgadd -d . -R /export/opt/client3 SUNWstuf
```

installs the objects in /export/opt/client3/opt/SUNWstuf; but that is the closest this package comes to being relocatable.

Notice the use of the question mark (?) for the /opt directory in the pkgmap file. This indicates that the existing attributes should not be changed. It does not mean "create the directory with default attributes," although under certain circumstances that may happen. Any directory that is specific to the new package must specify all attributes explicitly.

Composite Packages

Any package containing relocatable objects is referred to as a relocatable package. This can be misleading because a relocatable package may contain absolute paths in its pkgmap file. Using a root (/) entry in a pkgmap file can enhance the relocatable aspects of the package. Packages that have both relocatable and root entries are called *composite* packages.

Example-A Traditional Solution

Assume that one object in the SUNWstuf package is a startup script executed at run level 2. The file /etc/rc2.d/S70dostuf needs to be installed as a part of the package, but it cannot be placed into the base directory. Assuming that a relocatable package is the only solution, the pkginfo and a pkgmap might look like this.

The `pkginfo` *File*

```
# pkginfo file
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none
PSTAMP=hubert990707141632
```

The `pkgmap` *File*

```
: 1 1758
1 d none opt/SUNWstuf/EZstuf 0775 root bin
1 f none opt/SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none opt/SUNWstuf/HRDstuf 0775 root bin
1 f none opt/SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none opt/SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none etc ? ? ?
1 d none etc/rc2.d ? ? ?
1 f none etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
```

There is not much difference between this approach and that of the absolute package. In fact, this would be better off as an absolute package—if the administrator provided an alternate base directory for this package, it would not work!

In fact, only one file in this package needs to be root-relative, the rest could be moved anywhere. How to solve this problem through the use of a composite package is discussed throughout the remainder of this section.

Beyond Tradition

The approach described in this section does not apply to all packages, but it does result in improved performance during installation to an heterogeneous environment. Very little of this applies to packages that are delivered as part of the Solaris operating environment (bundled packages); however, unbundled packages can practice non-traditional packaging.

The reason behind encouraging relocatable packages is to support this requirement:

When a package is added or removed, the existing desirable behaviors of installed software products will be unchanged.

Unbundled packages should reside under `/opt` so as to assure that the new package does not interfere with existing products.

Another Look at Composite Packages

There are two rules to follow when constructing a functional composite package:

- Establish the base directory based upon where the vast majority of the package objects go.

- If a package object goes into a common directory that is not the base directory (for example, `/etc`), specify it as an absolute path name in the `prototype` file.

In other words, since "relocatable" means the object can be installed anywhere and still work, no startup script run by `init` at boot time can be considered relocatable! While there is nothing wrong with specifying `/etc/passwd` as a relative path in the delivered package, there is only one place it can go.

Making Absolute Path Names Look Relocatable

If you are going to construct a composite package, the absolute paths must operate in a manner which does not interfere with existing installed software. A package that can be entirely contained in `/opt` gets around this problem since there are no existing files in the way. When a file in `/etc` is included in the package, you must ensure that the absolute path names behave in the same way that is expected from relative path names. Consider the following two examples.

Example—Modifying a File

Description

An entry is being added to a table, or the object is a new table which is likely to be modified by other programs or packages.

Implementation

Define the object as file type `e` and belonging to the `build`, `awk`, or `sed` class. The script that performs this task must remove itself as effectively as it adds itself.

Example

An entry needs to be added to `/etc/vfstab` in support of the new solid state hard disk.

The entry in the `pkgmap` file might be

```
1 e sed /etc/vfstab ? ? ?
```

The `request` script asks the operator if `/etc/vfstab` should be modified by the package. If the operator answers "no" then the request script will print instructions on how to do the job manually and will execute

```
echo "CLASSES=none" >> $1
```

If the operator answers "yes" then it executes

```
echo "CLASSES=none sed" >> $1
```

which activates the class action script that will make the necessary modifications. The `sed` class means that the package file `/etc/vfstab` is a `sed` program which contains both the install and remove operations for the same-named file on the target system.

Example—Creating a New File

Description

The object is an entirely new file that is unlikely to be edited at a later time or, it is replacing a file owned by another package.

Implementation

Define the package object as file type `f` and install it using a class action script capable of undoing the change.

Example

A brand new file is required in `/etc` to provide the necessary information to
support the solid state hard disk, named `/etc/shdisk.conf`. The entry in the
`pkgmap` file might look like this:

```
.
.
.
1 f newetc /etc/shdisk.conf
.
.
.
```

The class action script `i.newetc` is responsible for installing this and any other files
that need to go into `/etc`. It checks to make sure there is not another file there. If
there is not, it will simply copy the new file into place. If there is already a file in
place, it will back it up before installing the new file. The script `r.newetc` removes
these files and restores the originals, if required. Here is the key fragment of the
install script.

```
# i.newetc
while read src dst; do
   if [ -f $dst ]; then
     dstfile=`basename $dst`
     cp $dst $PKGSAV/$dstfile
   fi
   cp $src $dst
done

if [ "${1}" = "ENDOFCLASS" ]; then
   cd $PKGSAV
   tar cf SAVE.newetc .
   $INST_DATADIR/$PKG/install/squish SAVE.newetc
fi
```

Notice that this script uses the `PKGSAV` environment variable to store a backup of the
file to be replaced. When the argument `ENDOFCLASS` is passed to the script, that is
the `pkgadd` command informing the script that these are the last entries in this class,
at which point the script archives and compresses the files that were saved using a
private compression program stored in the install directory of the package.

While the use of the `PKGSAV` environment variable is not reliable during a package
update; if the package is not updated (through a patch, for instance) the backup file
is secure. The following remove script includes code to deal with the other
issue—the fact that older versions of the `pkgrm` command do not pass the scripts the
correct path to the `PKGSAV` environment variable.

The removal script might look like this.

```
# r.newetc

# make sure we have the correct PKGSAV
if [ -d $PKG_INSTALL_ROOT$PKGSAV ]; then
    PKGSAV="$PKG_INSTALL_ROOT$PKGSAV"
fi

# find the unsquish program
UNSQUISH_CMD=`dirname $0`/unsquish

while read file; do
   rm $file
done

if [ "${1}" = ENDOFCLASS ]; then
    if [ -f $PKGSAV/SAVE.newetc.sq ]; then
        $UNSQUISH_CMD $PKGSAV/SAVE.newetc
    fi

    if [ -f $PKGSAV/SAVE.newetc ]; then
        targetdir=dirname $file # get the right directory
        cd $targetdir
        tar xf $PKGSAV/SAVE.newetc
        rm $PKGSAV/SAVE.newetc
    fi
fi
```

This script uses a private uninstalled algorithm (unsquish) which is in the install directory of the package database. This is done automatically by the pkgadd command at install time. All scripts not specifically recognized as install-only by the pkgadd command are left in this directory for use by the pkgrm command. You cannot count on where that directory is, but you can depend on the directory being flat and containing all appropriate information files and installation scripts for the package. This script finds the directory by virtue of the fact that the class action script is guaranteed to be executing from the directory that contains the unsquish program.

Notice, also, that this script does not just assume the target directory is /etc. It may actually be /export/root/client2/etc. The correct directory could be constructed in one of two ways.

- Use the ${PKG_INSTALL_ROOT}/etc construction, or

- Take the directory name of a file passed by the pkgadd command (which is what this script does).

By using this approach for each absolute object in the package, you can be sure that the current desirable behavior is unchanged or at least recoverable.

Example—A Composite Package

This is an example of the `pkginfo` and `pkgmap` files for a composite package.

The `pkginfo` *File*

```
PKG=SUNWstuf
NAME=software stuff
ARCH=sparc
VERSION=1.0.0,REV=1.0.5
CATEGORY=application
DESC=a set of utilities that do stuff
BASEDIR=/opt
VENDOR=Sun Microsystems, Inc.
HOTLINE=Please contact your local service provider
EMAIL=
MAXINST=1000
CLASSES=none daemon
PSTAMP=hubert990707141632
```

The `pkgmap` *File*

```
: 1 1758
1 d none SUNWstuf/EZstuf 0775 root bin
1 f none SUNWstuf/EZstuf/dirdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/usrdel 0555 bin bin 40 773 751310229
1 f none SUNWstuf/EZstuf/filedel 0555 bin bin 40 773 751310229
1 d none SUNWstuf/HRDstuf 0775 root bin
1 f none SUNWstuf/HRDstuf/mksmart 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mktall 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkcute 0555 bin bin 40 773 751310229
1 f none SUNWstuf/HRDstuf/mkeasy 0555 bin bin 40 773 751310229
1 d none /etc ? ? ?
1 d none /etc/rc2.d ? ? ?
1 e daemon /etc/rc2.d/S70dostuf 0744 root sys 450 223443
1 i i.daemon 509 39560 752978103
1 i pkginfo 348 28411 760740163
1 i postinstall 323 26475 751309908
1 i postremove 402 33179 751309945
1 i preinstall 321 26254 751310019
1 i preremove 320 26114 751309865
1 i r.daemon 320 24573 742152591
```

While `S70dostuf` belongs to the `daemon` class, the directories that lead up to it
(which are already in place at install time) belong to the `none` class. Even if the
directories were unique to this package, you should leave them in the `none` class.
The reason for this is that the directories need to be created first and deleted last, and

this is always true for the `none` class. The `pkgadd` command creates directories; they are not copied from the package and they are not passed to a class action script to be created. Instead, they are created by the `pkgadd` command before it calls the install class action script, and the `pkgrm` command deletes directories after completion of the removal class action script.

This means that if a directory in a special class contains objects in the class `none`, when the `pkgrm` command attempts to remove the directory, it fails because the directory will not be empty in time. If an object of class `none` is to be inserted into a directory of some special class, that directory will not exist in time to accept the object. The `pkgadd` command will create the directory on-the-fly during installation of the object and may not be able to synchronize the attributes of that directory when it finally sees the `pkgmap` definition.

Note - When assigning a directory to a class, always remember the order of creation and deletion.

Making Packages Installable Remotely

All packages *must* be installable remotely. Installable remotely means you do not assume the administrator installing your package is installing to the root (`/`) file system of the system running the `pkgadd` command. If, in one of your procedure scripts, you need to get to the `/etc/vfstab` file of the target system, you need to use the `PKG_INSTALL_ROOT` environment variable. In other words, the path name `/etc/vfstab` will get you to the `/etc/vfstab` file of the system running the `pkgadd` command, but the administrator may be installing to a client at `/export/root/client3`. The path `${PKG_INSTALL_ROOT}/etc/vfstab` is guaranteed to get you to the target file system.

Example—Installing to a Client

In this example, the `SUNWstuf` package is installed to `client3`, which is configured with `/opt` in its root (`/`) file system. One other version of this package is already installed on `client3`, and the base directory is set to `basedir=/opt/$PKGINST` from an administration file, `thisadmin`. (For more information on administration files, see "The Administrative Defaults File" on page 120.) The `pkgadd` command executed on the server is:

```
# pkgadd -a thisadmin -R /export/root/client3 SUNWstuf
```

The table below lists the environment variables and their values that are passed to the procedure scripts.

TABLE 6–1 Values Passed to Procedure Scripts

Environment Variable	Value
PKGINST	SUNWstuf.2
PKG_INSTALL_ROOT	/export/root/client3
CLIENT_BASEDIR	/opt/SUNWstuf.2
BASEDIR	/export/root/client3/opt/SUNWstuf.2

Example—Installing to a Server or Standalone

To install to the server or a standalone system under the same circumstances as the previous example, the command is:

```
# pkgadd -a thisadmin SUNWstuf
```

The table below lists the environment variables and their values that are passed to the procedure scripts.

TABLE 6–2 Values Passed to Procedure Scripts

Environment Variable	Value
PKGINST	SUNWstuf.2
PKG_INSTALL_ROOT	Not defined.
CLIENT_BASEDIR	/opt/SUNWstuf.2
BASEDIR	/opt/SUNWstuf.2

Example—Mounting Shared File Systems

Assume that the `SUNWstuf` package creates and shares a file system on the server at `/export/SUNWstuf/share`. When the package is installed to the clients, their `/etc/vfstab` files need to be updated to mount this shared file system. This is a situation where you can use the `CLIENT_BASEDIR` variable.

The entry on the client needs to present the mount point with reference to the client's file system. This line should be constructed correctly whether the installation is from the server or from the client. Assume that the server's system name is `$SERVER`. You can go to `$PKG_INSTALL_ROOT/etc/vfstab` and, using the `sed` or `awk` commands, construct the following line for the client's `/etc/vfstab` file.

```
$SERVER:/export/SUNWstuf/share - $CLIENT_BASEDIR/usr nfs - yes ro
```

For example, for the server `universe` and the client system `client9`, the line in the client system's `/etc/vfstab` file would look like:

```
universe:/export/SUNWstuf/share - /opt/SUNWstuf.2/usr nfs - yes ro
```

Using these parameters correctly, the entry always mounts the client's file system, whether it is being constructed locally or from the server.

Patching Packages

A patch to a package is just a sparse package designed to overwrite certain files in the original. There is no real reason for shipping a sparse package except to save space on the delivery medium. You could also ship the entire original package with a few files changed, or provide access to the modified package over a network. As long as only those new files are actually different (the other files were not recompiled), the `pkgadd` command installs the differences. Review the following guidelines regarding patching packages.

- A patch must not change the intended delivered behavior of the package—it is *not* a mechanism for installing new features. A patch is used to repair objects installed on the system.

- If the system is complex enough, it is wise to establish a patch identification system which assures that no two patches replace the same file in an attempt to correct different aberrant behaviors. For instance, Sun patch base numbers are assigned mutually exclusive sets of files for which they are responsible.

- It is necessary to be able to back out a patch.

It is crucial that the version number of the patch package be the same as that of the original package. This is true because a patch *must not* add functionality. You should keep track of the patch status of the package using a separate `pkginfo` file entry of the form

```
PATCH=patch_number
```

If the package version is changed for a patch, you create another instance of the package and it becomes extremely difficult to manage the patched product. This method of progressive instance patching carried certain advantages in the early releases of the Solaris operating environment, but makes management of more complicated systems tedious.

As far as the packages that make up the Solaris operating environment are concerned, there should be only one copy of the package in the package database, although there may be multiple patched instances. In order to remove an object from an installed package (using the `removef` command) you need to figure out what instances own that file.

However, if your package (that is not part of the Solaris operating environment) needs to determine the patch level of a particular package that *is* part of the Solaris operating environment, this becomes a problem to be resolved here. The installation scripts can be quite large without significant impact since they are not stored on the target file system. Using class action scripts and various other procedure scripts, you can save changed files using the `PKGSAV` environment variable (or to some other, more permanent directory) in order to allow backing out installed patches. You can also monitor patch history by setting appropriate environment variables through the `request` scripts. The scripts in the next sections assume that there may be multiple patches whose numbering scheme carries some meaning when applied to a single package. In this case, individual patch numbers represent a subset of functionally related files within the package. Two different patch numbers cannot change the same file.

In order to make a regular sparse package into a patch package, the scripts described in the following sections can simply be folded into the package. All of them are recognizable as standard package components with the exception of the last two which are named `patch_checkinstall` and `patch_postinstall`. Those two scripts can be incorporated into the backout package, if you want to include the ability to back out the patch. The scripts are fairly simple and their various tasks are straightforward.

Note - This method of patching can be used to patch client systems, but client root directories on the server must have the correct permissions to allow reading by the user `install` or `nobody`.

The `checkinstall` Script

The `checkinstall` script verifies that the patch is appropriate for this particular package. Once that is confirmed, it constructs the *patch list* and the *patch info* list, and then inserts them into the response file for incorporation into the package database.

A patch list is the list of patches that have affected the current package. This list of patches is recorded in the installed package in the pkginfo file with a line that might look like this:

```
PATCHLIST=patch_id patch_id ...
```

A patch info list is the list of patches on which the current patch is dependent. This list of patches is also recorded in the pkginfo file with a line that might look like this.

```
PATCH_INFO_103203-01=Installed... Obsoletes:103201-01 Requires: \ Incompatibles: 120134-01
```

Note - These lines (and their format) are declared as a public interface. Any company that ships patches for Solaris packages should update this list appropriately. When a patch is delivered, each package within the patch contains a checkinstall script that performs this task. That same checkinstall script also updates some other patch-specific parameters. This is the new patch architecture, which is called Direct Instance Patching.

In this example, both the original packages and their patches exist in the same directory. The two original packages are named SUNWstuf.v1 and SUNWstuf.v2, and their patches are named SUNWstuf.p1 and SUNWstuf.p2. What this means is that it could be very difficult for a procedure script to figure out what directory these files came from, since everything in the package name after the dot (".") is stripped for the PKG parameter, and the PKGINST environment variable refers to the installed instance not the source instance. So the procedure scripts can find the source directory, the checkinstall script (which is always executed from the source directory) makes the inquiry and passes the location on as the variable SCRIPTS_DIR. If there had been only one package in the source directory called SUNWstuf, then the procedure scripts could have found it using $INSTDIR/$PKG.

```
# checkinstall script to control a patch installation.
# directory format options.
#
#       @(#)checkinstall 1.6 96/09/27 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH

INFO_DIR=`dirname $0`
INFO_DIR=`dirname $INFO_DIR`     # one level up

NOVERS_MSG="PaTcH_MsG 8 Version $VERSION of $PKG is not installed on this system."
```

(continued)

```
ALRDY_MSG="PaTcH_MsG 2 Patch number $Patch_label is already applied."
TEMP_MSG="PaTcH_MsG 23 Patch number $Patch_label cannot be applied until all \
restricted patches are backed out."

# Read the provided environment from what may have been a request script
. $1

# Old systems can't deal with checkinstall scripts anyway
if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        exit 0
fi


#
# Confirm that the intended version is installed on the system.
#
if [ "${UPDATE}" != "yes" ]; then
        echo "$NOVERS_MSG"
        exit 3
fi


#
# Confirm that this patch hasn't already been applied and
# that no other mix-ups have occurred involving patch versions and
# the like.
#
Skip=0
active_base=`echo $Patch_label | nawk '
        { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`
active_inst=`echo $Patch_label | nawk '
        { print substr($0, match($0, "Patchvers_pfx")+Patchvers_pfx_lnth) } '`

# Is this a restricted patch?
if echo $active_base | egrep -s "Patchstrict_str"; then
        is_restricted="true"
        # All restricted patches are backoutable
        echo "PATCH_NO_UNDO=" >> $1
else
        is_restricted="false"
fi

for patchappl in ${PATCHLIST}; do
        # Is this an ordinary patch applying over a restricted patch?
        if [ $is_restricted = "false" ]; then
                if echo $patchappl | egrep -s "Patchstrict_str"; then
                        echo "$TEMP_MSG"
                        exit 3;
                fi
        fi

        # Is there a newer version of this patch?
        appl_base=`echo $patchappl | nawk '
                { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`
        if [ $appl_base = $active_base ]; then
                appl_inst=`echo $patchappl | nawk '
```

```
                        { print substr($0, match($0, "Patchvers_pfx")\
+Patchvers_pfx_lnth) } ''
                result='expr $appl_inst \> $active_inst'
                if [ $result -eq 1 ]; then
                        echo "PaTcH_MsG 1 Patch number $Patch_label is \
superceded by the already applied $patchappl."
                        exit 3
                elif [ $appl_inst = $active_inst ]; then
                        # Not newer, it's the same
                        if [ "$PATCH_UNCONDITIONAL" = "true" ]; then
                                if [ -d $PKGSAV/$Patch_label ]; then
                                        echo "PATCH_NO_UNDO=true" >> $1
                                fi
                        else
                                echo "$ALRDY_MSG"
                                exit 3;
                        fi
                fi
        fi
done

# Construct a list of applied patches in order
echo "PATCHLIST=${PATCHLIST} $Patch_label" >> $1

#
# Construct the complete list of patches this one obsoletes
#
ACTIVE_OBSOLETES=$Obsoletes_label

if [ -n "$Obsoletes_label" ]; then
        # Merge the two lists
        echo $Obsoletes_label | sed 'y/\ /\n/' | \
        nawk -v PatchObsList="$PATCH_OBSOLETES" '
        BEGIN {
                printf("PATCH_OBSOLETES=");
                PatchCount=split(PatchObsList, PatchObsComp, " ");

                for(PatchIndex in PatchObsComp) {
                        Atisat=match(PatchObsComp[PatchIndex], "@");
                        PatchObs[PatchIndex]=substr(PatchObsComp[PatchIndex], \
0, Atisat-1);
                        PatchObsCnt[PatchIndex]=substr(PatchObsComp\
[PatchIndex], Atisat+1);
                }
        }
        {
                Inserted=0;
                for(PatchIndex in PatchObs) {
                        if (PatchObs[PatchIndex] == $0) {
                                if (Inserted == 0) {
                                        PatchObsCnt[PatchIndex]=PatchObsCnt\
[PatchIndex]+1;

                                        Inserted=1;
                                } else {
```

(continued)

```
                                    PatchObsCnt[PatchIndex]=0;
                            }
                    }
            }
            if (Inserted == 0) {
                    printf ("%s@1 ", $0);
            }
            next;
        }
        END {
                for(PatchIndex in PatchObs) {
                        if ( PatchObsCnt[PatchIndex] != 0) {
                                printf("%s@%d ", PatchObs[PatchIndex], \
PatchObsCnt[PatchIndex]);
                        }
                }
                printf("\n");
        } ' >> $1
        # Clear the parameter since it has already been used.
        echo "Obsoletes_label=" >> $1

        # Pass it's value on to the preinstall under another name
        echo "ACTIVE_OBSOLETES=$ACTIVE_OBSOLETES" >> $1
fi

#
# Construct PATCH_INFO line for this package.
#

tmpRequire=`nawk -F= ' $1 ~ /REQUIR/ { print $2 } ' $INFO_DIR/pkginfo `
tmpIncompat=`nawk -F= ' $1 ~ /INCOMPAT/ { print $2 } ' $INFO_DIR/pkginfo `

if [ -n "$tmpRequire" ] && [ -n "$tmpIncompat" ]
then
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: $tmpRequire \
          Incompatibles: $tmpIncompat" >> $1
elif [ -n "$tmpRequire" ]
then
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: $tmpRequire \
Incompatibles: " >> $1
elif [ -n "$tmpIncompat" ]
then
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: Incompatibles: \
$tmpIncompat" >> $1
else
        echo "PATCH_INFO_$Patch_label=Installed: `date` From: `uname -n` \
          Obsoletes: $ACTIVE_OBSOLETES Requires: Incompatibles: " >> $1
fi

#
# Since this script is called from the delivery medium and we may be using
```

(continued)

```
# dot extensions to distinguish the different patch packages, this is the
# only place we can, with certainty, trace that source for our backout
# scripts. (Usually $INST_DATADIR would get us there).
#
echo "SCRIPTS_DIR=`dirname $0`" >> $1

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

exit 0
```

The `preinstall` Script

The `preinstall` script initializes the `prototype` file, information files, and
installation scripts for the backout package to be constructed. This script is very
simple and the remaining scripts in this example only allow a backout package to
describe regular files.

If you wanted to restore symbolic links, hard links, devices, and named pipes in a
backout package, you could modify the `preinstall` script to use the `pkgproto`
command to compare the delivered `pkgmap` file with the installed files, and then
create a `prototype` file entry for each non-file to be changed in the backout package.
The method you should use is similar to the method in the class action script.

The scripts `patch_checkinstall` and `patch_postinstall` are inserted into the
package source tree from the `preinstall` script. These two scripts undo what the
patch does.

```
# This script initializes the backout data for a patch package
# directory format options.
#
#        @(#)preinstall 1.5 96/05/10 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH
recovery="no"

if [ "$PKG_INSTALL_ROOT" = "/" ]; then
        PKG_INSTALL_ROOT=""
fi
```

(continued)

```
# Check to see if this is a patch installation retry.
if [ "$INTERRUPTION" = "yes" ]; then
    if [ -d "$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST" ] || [ -d \
"$PATCH_BUILD_DIR/$Patch_label.$PKGINST" ]; then
        recovery="yes"
    fi
fi

if [ -n "$PATCH_BUILD_DIR" -a -d "$PATCH_BUILD_DIR" ]; then
        BUILD_DIR="$PATCH_BUILD_DIR/$Patch_label.$PKGINST"
else
        BUILD_DIR="$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST"
fi

FILE_DIR=$BUILD_DIR/files
RELOC_DIR=$BUILD_DIR/files/reloc
ROOT_DIR=$BUILD_DIR/files/root
PROTO_FILE=$BUILD_DIR/prototype
PKGINFO_FILE=$BUILD_DIR/pkginfo
THIS_DIR=`dirname $0`

if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        # If this is being used in an old-style patch, insert
        # the old-style script commands here.

        #XXXOld_CommandsXXX#

        exit 0
fi

#
# Unless specifically denied, initialize the backout patch data by
# creating the build directory and copying over the original pkginfo
# which pkgadd saved in case it had to be restored.
#
if [ "$PATCH_NO_UNDO" != "true" ] && [ "$recovery" = "no" ]; then
        if [ -d $BUILD_DIR ]; then
                rm -r $BUILD_DIR
        fi

        # If this is a retry of the same patch then recovery is set to
        # yes. Which means there is a build directory already in
        # place with the correct backout data.

        if [ "$recovery" = "no" ]; then
                mkdir $BUILD_DIR
                mkdir -p $RELOC_DIR
                mkdir $ROOT_DIR
        fi

        #
        # Here we initialize the backout pkginfo file by first
        # copying over the old pkginfo file and themn adding the
```

(continued)

```
        # ACTIVE_PATCH parameter so the backout will know what patch
        # it's backing out.
        #
        # NOTE : Within the installation, pkgparam returns the
        # original data.
        #
        pkgparam -v $PKGINST | nawk '
                $1 ~ /PATCHLIST/        { next; }
                $1 ~ /PATCH_OBSOLETES/  { next; }
                $1 ~ /ACTIVE_OBSOLETES/ { next; }
                $1 ~ /Obsoletes_label/  { next; }
                $1 ~ /ACTIVE_PATCH/     { next; }
                $1 ~ /Patch_label/      { next; }
                $1 ~ /UPDATE/    { next; }
                $1 ~ /SCRIPTS_DIR/      { next; }
                $1 ~ /PATCH_NO_UNDO/    { next; }
                $1 ~ /INSTDATE/ { next; }
                $1 ~ /PKGINST/  { next; }
                $1 ~ /OAMBASE/  { next; }
                $1 ~ /PATH/     { next; }
                { print; } ' > $PKGINFO_FILE
        echo "ACTIVE_PATCH=$Patch_label" >> $PKGINFO_FILE
        echo "ACTIVE_OBSOLETES=$ACTIVE_OBSOLETES" >> $PKGINFO_FILE

        # And now initialize the backout prototype file with the
        # pkginfo file just formulated.
        echo "i pkginfo" > $PROTO_FILE

        # Copy over the backout scripts including the undo class
        # action scripts
        for script in $SCRIPTS_DIR/*; do
                srcscript=`basename $script`
                targscript=`echo $srcscript | nawk '
                        { script=$0; }
                        /u\./ {
                                sub("u.", "i.", script);
                                print script;
                                next;
                        }
                        /patch_/ {
                                sub("patch_", "", script);
                                print script;
                                next;
                        }
                        { print "dont_use" } '`
                if [ "$targscript" = "dont_use" ]; then
                        continue
                fi

                echo "i $targscript=$FILE_DIR/$targscript" >> $PROTO_FILE
                cp $SCRIPTS_DIR/$srcscript $FILE_DIR/$targscript
        done
        #
        # Now add entries to the prototype file that won't be passed to
```

```
        # class action scripts. If the entry is brand new, add it to the
        # deletes file for the backout package.
        #
        Our_Pkgmap=`dirname $SCRIPTS_DIR`/pkgmap
        BO_Deletes=$FILE_DIR/deletes

        nawk -v basedir=${BASEDIR:-/} '
                BEGIN { count=0; }
                {
                        token = $2;
                        ftype = $1;
                }
                $1 ~ /[#\!:]/ { next; }
                $1 ~ /[0123456789]/ {
                        if ( NF >= 3) {
                                token = $3;
                                ftype = $2;
                        } else {
                                next;
                        }
                }
                { if (ftype == "i" || ftype == "e" || ftype == "f" || ftype == \
"v" || ftype == "d") { next; } }
                {
                        equals=match($4, "=")-1;
                        if ( equals == -1 ) { print $3, $4; }
                        else { print $3, substr($4, 0, equals); }
                }
                ' < $Our_Pkgmap | while read class path; do
                        #
                        # NOTE: If pkgproto is passed a file that is
                        # actually a hard link to another file, it
                        # will return ftype "f" because the first link
                        # in the list (consisting of only one file) is
                        # viewed by pkgproto as the source and always
                        # gets ftype "f".
                        #
                        # If this isn't replacing something, then it
                        # just goes to the deletes list.
                        #
                        if valpath -l $path; then
                                Chk_Path="$BASEDIR/$path"
                                Build_Path="$RELOC_DIR/$path"
                                Proto_From="$BASEDIR"
                        else    # It's an absolute path
                                Chk_Path="$PKG_INSTALL_ROOT$path"
                                Build_Path="$ROOT_DIR$path"
                                Proto_From="$PKG_INSTALL_ROOT"
                        fi
                         #
                        # Hard links have to be restored as regular files.
                        # Unlike the others in this group, an actual
                        # object will be required for the pkgmk.
                        #
```

(continued)

```
                            if [ -f "$Chk_Path" ]; then
                                    mkdir -p `dirname $Build_Path`
                                    cp $Chk_Path $Build_Path
                                    cd $Proto_From
                                    pkgproto -c $class "$Build_Path=$path" 1>> \
$PROTO_FILE 2> /dev/null
                                    cd $THIS_DIR
                            elif [ -h "$Chk_Path" -o \
                                  -c "$Chk_Path" -o \
                                  -b "$Chk_Path" -o \
                                  -p "$Chk_Path" ]; then
                                    pkgproto -c $class "$Chk_Path=$path" 1>> \
$PROTO_FILE 2> /dev/null
                            else
                                    echo $path >> $BO_Deletes
                            fi
                done
fi

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

exit 0
```

The Class Action Script

The class action script creates a copy of each file that replaces an existing file and
adds a corresponding line to the prototype file for the backout package. This is all
done with fairly simple nawk scripts. The class action script receives a list of source/
destination pairs consisting of ordinary files that do not match the corresponding
installed files. Symbolic links and other non-files must be dealt with in the
preinstall script.

```
# This class action script copies the files being replaced
# into a package being constructed in $BUILD_DIR. This class
# action script is only appropriate for regular files that
# are installed by simply copying them into place.
#
# For special package objects such as editable files, the patch
# producer must supply appropriate class action scripts.
#
# directory format options.
#
#       @(#)i.script 1.6 96/05/10 SMI
```

(continued)

```
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH

ECHO="/usr/bin/echo"
SED="/usr/bin/sed"
PKGPROTO="/usr/bin/pkgproto"
EXPR="/usr/bin/expr"    # used by dirname
MKDIR="/usr/bin/mkdir"
CP="/usr/bin/cp"
RM="/usr/bin/rm"
MV="/usr/bin/mv"

recovery="no"
Pn=$$
procIdCtr=0

CMDS_USED="$ECHO $SED $PKGPROTO $EXPR $MKDIR $CP $RM $MV"
LIBS_USED=""

if [ "$PKG_INSTALL_ROOT" = "/" ]; then
        PKG_INSTALL_ROOT=""
fi

# Check to see if this is a patch installation retry.
if [ "$INTERRUPTION" = "yes" ]; then
        if [ -d "$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST" ] ||
\
[ -d "$PATCH_BUILD_DIR/$Patch_label.$PKGINST" ]; then
                recovery="yes"
        fi
fi

if [ -n "$PATCH_BUILD_DIR" -a -d "$PATCH_BUILD_DIR" ]; then
        BUILD_DIR="$PATCH_BUILD_DIR/$Patch_label.$PKGINST"
else
        BUILD_DIR="$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST"
fi

FILE_DIR=$BUILD_DIR/files
RELOC_DIR=$FILE_DIR/reloc
ROOT_DIR=$FILE_DIR/root
BO_Deletes=$FILE_DIR/deletes
PROGNAME=`basename $0`

if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        PATCH_NO_UNDO="true"
fi

# Since this is generic, figure out the class.
Class=`echo $PROGNAME | nawk ' { print substr($0, 3)  }'`
```

(continued)

```
# Since this is an update, $BASEDIR is guaranteed to be correct
BD=${BASEDIR:-/}

cd $BD

#
# First, figure out the dynamic libraries that can trip us up.
#
if [ -z "$PKG_INSTALL_ROOT" ]; then
        if [ -x /usr/bin/ldd ]; then
                LIB_LIST='/usr/bin/ldd $CMDS_USED | sort -u | nawk '
                        $1 ~ /\// { continue; }
                        { printf "%s ", $3 } ''
        else
                LIB_LIST="/usr/lib/libc.so.1 /usr/lib/libdl.so.1
\
/usr/lib/libw.so.1 /usr/lib/libintl.so.1 /usr/lib/libadm.so.1 \
/usr/lib/libelf.so.1"
        fi
fi


#
# Now read the list of files in this class to be replaced. If the file
# is already in place, then this is a change and we need to copy it
# over to the build directory if undo is allowed. If it's a new entry
# (No $dst), then it goes in the deletes file for the backout package.
#
procIdCtr=0
while read src dst; do
        if [ -z "$PKG_INSTALL_ROOT" ]; then
                Chk_Path=$dst
                for library in $LIB_LIST; do
                        if [ $Chk_Path = $library ]; then
                                $CP $dst $dst.$Pn
                                LIBS_USED="$LIBS_USED $dst.$Pn"
                                LD_PRELOAD="$LIBS_USED"
                                export LD_PRELOAD
                        fi
                done
        fi

        if [ "$PATCH_PROGRESSIVE" = "true" ]; then
                # If this is being used in an old-style patch, insert
                # the old-style script commands here.

                #XXXOld_CommandsXXX#
                echo >/dev/null # dummy
        fi

        if [ "${PATCH_NO_UNDO}" != "true" ]; then
                #
                # Here we construct the path to the appropriate source
                # tree for the build. First we try to strip BASEDIR. If
```

```
                # there's no BASEDIR in the path, we presume that it is
                # absolute and construct the target as an absolute path
                # by stripping PKG_INSTALL_ROOT. FS_Path is the path to
                # the file on the file system (for deletion purposes).
                # Build_Path is the path to the object in the build
                # environment.
                #
                if [ "$BD" = "/" ]; then
                        FS_Path=`$ECHO $dst | $SED s@"$BD"@@`
                else
                        FS_Path=`$ECHO $dst | $SED s@"$BD/"@@`
                fi

                # If it's an absolute path the attempt to strip the
                # BASEDIR will have failed.
                if [ $dst = $FS_Path ]; then
                        if [ -z "$PKG_INSTALL_ROOT" ]; then
                                FS_Path=$dst
                                Build_Path="$ROOT_DIR$dst"
                        else
                                Build_Path="$ROOT_DIR`echo $dst | \
                                    sed s@"$PKG_INSTALL_ROOT"@@`"
                                FS_Path=`echo $dst | \
                                    sed s@"$PKG_INSTALL_ROOT"@@`
                        fi
                else
                        Build_Path="$RELOC_DIR/$FS_Path"
                fi

                if [ -f $dst ]; then     # If this is replacing something
                        cd $FILE_DIR
                        #
                        # Construct the prototype file entry. We replace
                        # the pointer to the filesystem object with the
                        # build directory object.
                        #
                        $PKGPROTO -c $Class $dst=$FS_Path | \
                            $SED -e s@=$dst@=$Build_Path@ >> \
                            $BUILD_DIR/prototype

                        # Now copy over the file
                        if [ "$recovery" = "no" ]; then
                                DirName=`dirname $Build_Path`
                                $MKDIR -p $DirName
                                $CP -p $dst $Build_Path
                        else
                                # If this file is already in the build area skip it
                                if [ -f "$Build_Path" ]; then
                                        cd $BD
                                        continue
                                else
                                        DirName=`dirname $Build_Path`
                                        if [ ! -d "$DirName" ]; then
                                                $MKDIR -p $DirName
```

(continued)

```
                                            fi
                                            $CP -p $dst $Build_Path
                                fi
                        fi

                        cd $BD
                else    # It's brand new
                        $ECHO $FS_Path >> $BO_Deletes
                fi
        fi

        # If special processing is required for each src/dst pair,
        # add that here.
        #
        #XXXSpecial_CommandsXXX#
        #

        $CP $src $dst.$$$procIdCtr
        if [ $? -ne 0 ]; then
                $RM $dst.$$$procIdCtr 1>/dev/null 2>&1
        else
                $MV -f $dst.$$$procIdCtr $dst
                for library in $LIB_LIST; do
                        if [ "$library" = "$dst" ]; then
                                LD_PRELOAD="$dst"
                                export LD_PRELOAD
                        fi
                done
        fi
        procIdCtr=`expr $procIdCtr + 1`
done

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

#
# Release the dynamic libraries
#
for library in $LIBS_USED; do
        $RM -f $library
done

exit 0
```

The `postinstall` Script

The `postinstall` script creates the backout package using the information provided by the other scripts. Since the `pkgmk` and `pkgtrans` commands do not require the package database, they can be executed within a package installation.

In the example, undoing the patch is permitted by constructing a stream format package in the save directory (using the `PKGSAV` environment variable). It is not obvious, but this package must be in stream format, because the save directory gets moved around during a `pkgadd` operation. If the `pkgadd` command is applied to a package in its own save directory, assumptions about where the package source is at any given time become very unreliable. A stream format package is unpacked into a temporary directory and installed from there. (A directory format package would begin installing from the save directory and find itself suddenly relocated during a `pkgadd` fail-safe operation.)

To determine which patches are applied to a package, use this command:

```
$ pkgparam SUNWstuf PATCHLIST
```

With the exception of `PATCHLIST`, which is a Sun public interface, there is nothing significant in the parameter names in this example. Instead of `PATCH` you could use the traditional `SUNW_PATCHID` and the various other lists such as `PATCH_EXCL` and `PATCH_REQD` could be renamed accordingly.

If certain patch packages depend upon other patch packages which are available from the same medium, the `checkinstall` script could determine this and create a script to be executed by the `postinstall` script in the same way that the upgrade example (see "Upgrading Packages" on page 168) does.

```
# This script creates the backout package for a patch package
#
# directory format options.
#
# @(#) postinstall 1.6 96/01/29 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

# Description:
#       Set the TYPE parameter for the remote file
#
# Parameters:
#       none
#
# Globals set:
#       TYPE

set_TYPE_parameter () {
        if [ ${PATCH_UNDO_ARCHIVE:?????} = "/dev" ]; then
                # handle device specific stuff
```

(continued)

```
                TYPE="removable"
        else
                TYPE="filesystem"
        fi
}


#
# Description:
#       Build the remote file that points to the backout data
#
# Parameters:
#       $1:     the un/compressed undo archive
#
# Globals set:
#       UNDO, STATE

build_remote_file () {
        remote_path=$PKGSAV/$Patch_label/remote
        set_TYPE_parameter
        STATE="active"

        if [ $1 = "undo" ]; then
                UNDO="undo"
        else
                UNDO="undo.Z"
        fi

        cat > $remote_path << EOF
# Backout data stored remotely
TYPE=$TYPE
FIND_AT=$ARCHIVE_DIR/$UNDO
STATE=$STATE
EOF
}

PATH=/usr/sadm/bin:$PATH

if [ "$PKG_INSTALL_ROOT" = "/" ]; then
        PKG_INSTALL_ROOT=""
fi

if [ -n "$PATCH_BUILD_DIR" -a -d "$PATCH_BUILD_DIR" ]; then
        BUILD_DIR="$PATCH_BUILD_DIR/$Patch_label.$PKGINST"
else
        BUILD_DIR="$PKG_INSTALL_ROOT/var/tmp/$Patch_label.$PKGINST"
fi

if [ ! -n "$PATCH_UNDO_ARCHIVE" ]; then
        PATCH_UNDO_ARCHIVE="none"
fi

FILE_DIR=$BUILD_DIR/files
RELOC_DIR=$FILE_DIR/reloc
ROOT_DIR=$FILE_DIR/root
```

(continued)

Advanced Package Creation Techniques **161**

```
BO_Deletes=$FILE_DIR/deletes
THIS_DIR=`dirname $0`
PROTO_FILE=$BUILD_DIR/prototype
TEMP_REMOTE=$PKGSAV/$Patch_label/temp

if [ "$PATCH_PROGRESSIVE" = "true" ]; then
        # remove the scripts that are left behind
        install_scripts=`dirname $0`
        rm $install_scripts/checkinstall \
$install_scripts/patch_checkinstall $install_scripts/patch_postinstall

        # If this is being used in an old-style patch, insert
        # the old-style script commands here.

        #XXXOld_CommandsXXX#

        exit 0
fi
#
# At this point we either have a deletes file or we don't. If we do,
# we create a prototype entry.
#
if [ -f $BO_Deletes ]; then
        echo "i deletes=$BO_Deletes" >> $BUILD_DIR/prototype
fi


#
# Now delete everything in the deletes list after transferring
# the file to the backout package and the entry to the prototype
# file. Remember that the pkgmap will get the CLIENT_BASEDIR path
# but we have to actually get at it using the BASEDIR path. Also
# remember that removef will import our PKG_INSTALL_ROOT
#
Our_Deletes=$THIS_DIR/deletes
if [ -f $Our_Deletes ]; then
        cd $BASEDIR

        cat $Our_Deletes | while read path; do
                Reg_File=0

                if valpath -l $path; then
                        Client_Path="$CLIENT_BASEDIR/$path"
                        Build_Path="$RELOC_DIR/$path"
                        Proto_Path=$BASEDIR/$path
                else    # It's an absolute path
                        Client_Path=$path
                        Build_Path="$ROOT_DIR$path"
                        Proto_Path=$PKG_INSTALL_ROOT$path
                fi

                # Note: If the file isn't really there, pkgproto
                # doesn't write anything.
                LINE=`pkgproto $Proto_Path=$path`
                ftype=`echo $LINE | nawk '{ print $1 }'`
```

(continued)

```
                if [ $ftype = "f" ]; then
                        Reg_File=1
                fi

                if [ $Reg_File = 1 ]; then
                        # Add source file to the prototype entry
                        if [ "$Proto_Path" = "$path" ]; then
                                LINE=`echo $LINE | sed -e s@$Proto_Path@$Build_Path@2`
                        else
                                LINE=`echo $LINE | sed -e s@$Proto_Path@$Build_Path@`
                        fi

                        DirName=`dirname $Build_Path`
                        # make room in the build tree
                        mkdir -p $DirName
                        cp -p $Proto_Path $Build_Path
                fi

                # Insert it into the prototype file
                echo $LINE 1>>$PROTO_FILE 2>/dev/null

                # Remove the file only if it's OK'd by removef
                rm `removef $PKGINST $Client_Path` 1>/dev/null 2>&1
        done
        removef -f $PKGINST

        rm $Our_Deletes
fi

#
# Unless specifically denied, make the backout package.
#
if [ "$PATCH_NO_UNDO" != "true" ]; then
        cd $BUILD_DIR   # We have to build from here.

        if [ "$PATCH_UNDO_ARCHIVE" != "none" ]; then
                STAGE_DIR="$PATCH_UNDO_ARCHIVE"
                ARCHIVE_DIR="$PATCH_UNDO_ARCHIVE/$Patch_label/$PKGINST"
                mkdir -p $ARCHIVE_DIR
                mkdir -p $PKGSAV/$Patch_label
        else
                if [ -d $PKGSAV/$Patch_label ]; then
                        rm -r $PKGSAV/$Patch_label
                fi
                STAGE_DIR=$PKGSAV
                ARCHIVE_DIR=$PKGSAV/$Patch_label
                mkdir $ARCHIVE_DIR
        fi

        pkgmk -o -d $STAGE_DIR 1>/dev/null 2>&1
        pkgtrans -s $STAGE_DIR $ARCHIVE_DIR/undo $PKG 1>/dev/null 2>&1
        compress $ARCHIVE_DIR/undo
        retcode=$?
        if [ "$PATCH_UNDO_ARCHIVE" != "none" ]; then
```

(continued)

Advanced Package Creation Techniques   **163**

```
                    if [ $retcode != 0 ]; then
                            build_remote_file "undo"
                    else
                            build_remote_file "undo.Z"
                    fi
        fi
        rm -r $STAGE_DIR/$PKG

        cd ..
        rm -r $BUILD_DIR
        # remove the scripts that are left behind
        install_scripts=`dirname $0`
        rm $install_scripts/checkinstall $install_scripts/patch_\
checkinstall $install_scripts/patch_postinstall
fi

#
# Since this apparently worked, we'll mark as obsoleted the prior
# versions of this patch - installpatch deals with explicit obsoletions.
#
cd ${PKG_INSTALL_ROOT:-/}
cd var/sadm/pkg

active_base=`echo $Patch_label | nawk '
        { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`

List=`ls -d $PKGINST/save/${active_base}*`
if [ $? -ne 0 ]; then
        List=""
fi

for savedir in $List; do
        patch=`basename $savedir`
        if [ $patch = $Patch_label ]; then
                break
        fi

        # If we get here then the previous patch gets deleted
        if [ -f $savedir/undo ]; then
                mv $savedir/undo $savedir/obsolete
                echo $Patch_label >> $savedir/obsoleted_by
        elif [ -f $savedir/undo.Z ]; then
                mv $savedir/undo.Z $savedir/obsolete.Z
                echo $Patch_label >> $savedir/obsoleted_by
        elif  [ -f $savedir/remote ]; then
                `grep . $PKGSAV/$patch/remote | sed 's/STATE=.*/STATE=obsolete/
' > $TEMP_REMOTE`
                rm -f $PKGSAV/$patch/remote
                mv $TEMP_REMOTE $PKGSAV/$patch/remote
                rm -f $TEMP_REMOTE
                echo $Patch_label >> $savedir/obsoleted_by
        elif  [ -f $savedir/obsolete -o -f $savedir/obsolete.Z ]; then
                echo $Patch_label >> $savedir/obsoleted_by
        fi
```

(continued)

```
done

# If additional operations are required for this package, place
# those package-specific commands here.

#XXXSpecial_CommandsXXX#

exit 0
```

The patch_checkinstall
Script

```
# checkinstall script to validate backing out a patch.
# directory format option.
#
#        @(#)patch_checkinstall 1.2 95/10/10 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#

PATH=/usr/sadm/bin:$PATH

LATER_MSG="PaTcH_MsG 6 ERROR: A later version of this patch is applied."
NOPATCH_MSG="PaTcH_MsG 2 ERROR: Patch number $ACTIVE_PATCH is not installed"
NEW_LIST=""

# Get OLDLIST
. $1


#
# Confirm that the patch that got us here is the latest one installed on
# the system and remove it from PATCHLIST.
#
Is_Inst=0
Skip=0
active_base=`echo $ACTIVE_PATCH | nawk '
        { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`
active_inst=`echo $ACTIVE_PATCH | nawk '
        { print substr($0, match($0, "Patchvers_pfx")+1) } '`
for patchappl in ${OLDLIST}; do
        appl_base=`echo $patchappl | nawk '
                { print substr($0, 1, match($0, "Patchvers_pfx")-1) } '`
        if [ $appl_base = $active_base ]; then
                appl_inst=`echo $patchappl | nawk '
                        { print substr($0, match($0, "Patchvers_pfx")+1) } '`
```

(continued)

```
                 result='expr $appl_inst \> $active_inst'
                 if [ $result -eq 1 ]; then
                         puttext "$LATER_MSG"
                         exit 3
                 elif [ $appl_inst = $active_inst ]; then
                         Is_Inst=1
                         Skip=1
                 fi
        fi
        if [ $Skip = 1 ]; then
                 Skip=0
        else
                 NEW_LIST="${NEW_LIST} $patchappl"
        fi
done

if [ $Is_Inst = 0 ]; then
        puttext "$NOPATCH_MSG"
        exit 3
fi


#
# OK, all's well. Now condition the key variables.
#
echo "PATCHLIST=${NEW_LIST}" >> $1
echo "Patch_label=" >> $1
echo "PATCH_INFO_$ACTIVE_PATCH=backed out" >> $1

# Get the current PATCH_OBSOLETES and condition it
Old_Obsoletes=$PATCH_OBSOLETES

echo $ACTIVE_OBSOLETES | sed 'y/\ /\n/' | \
nawk -v PatchObsList="$Old_Obsoletes" '
        BEGIN {
                 printf("PATCH_OBSOLETES=");
                 PatchCount=split(PatchObsList, PatchObsComp, " ");

                 for(PatchIndex in PatchObsComp) {
                         Atisat=match(PatchObsComp[PatchIndex], "@");
                         PatchObs[PatchIndex]=substr(PatchObsComp[PatchIndex], \
0, Atisat-1);
                         PatchObsCnt[PatchIndex]=substr(PatchObsComp\
[PatchIndex], Atisat+1);
                 }
        }
        {
                 for(PatchIndex in PatchObs) {
                         if (PatchObs[PatchIndex] == $0) {
                                 PatchObsCnt[PatchIndex]=PatchObsCnt[PatchIndex]-1;
                         }
                 }
                 next;
        }
        END {
```

```
                for(PatchIndex in PatchObs) {
                        if ( PatchObsCnt[PatchIndex] > 0 ) {
                                printf("%s@%d ", PatchObs[PatchIndex], PatchObsCnt\
[PatchIndex]);
                        }
                }
                printf("\n");
        } ' >> $1

        # remove the used parameters
        echo "ACTIVE_OBSOLETES=" >> $1
        echo "Obsoletes_label=" >> $1

exit 0
```

The `patch_postinstall` Script

```
# This script deletes the used backout data for a patch package
# and removes the deletes file entries.
#
# directory format options.
#
#       @(#)patch_postinstall 1.2 96/01/29 SMI
#
# Copyright (c) 1995 by Sun Microsystems, Inc.
# All rights reserved
#
PATH=/usr/sadm/bin:$PATH
THIS_DIR='dirname $0'

Our_Deletes=$THIS_DIR/deletes

#
# Delete the used backout data
#
if [ -f $Our_Deletes ]; then
        cat $Our_Deletes | while read path; do
                if valpath -l $path; then
                        Client_Path='echo "$CLIENT_BASEDIR/$path" | sed s@//@/@'
                else    # It's an absolute path
                        Client_Path=$path
                fi
                rm 'removef $PKGINST $Client_Path'
        done
        removef -f $PKGINST
```

(continued)

Advanced Package Creation Techniques   **167**

```
        rm $Our_Deletes
fi

#
# Remove the deletes file, checkinstall and the postinstall
#
rm -r $PKGSAV/$ACTIVE_PATCH
rm -f $THIS_DIR/checkinstall $THIS_DIR/postinstall

exit 0
```

Upgrading Packages

The process of upgrading a package is very different from that of overwriting a package. While there are special tools to support the upgrade of standard packages delivered as part of the Solaris operating environment, an unbundled package can be designed to support its own upgrade—several previous examples described packages that look ahead and control the precise method of installation under the direction of the administrator. You can design the request script to support direct upgrade of a package as well. If the administrator chooses to have one package install so as to completely replace another, leaving no residual obsolete files, the package scripts can do this.

The request script and postinstall script in this example provide a simple upgradable package. The request script communicates with the administrator and then sets up a simple file in the /tmp directory to remove the old package instance. (Although the request script creates a file (which is forbidden), it is okay because everyone has access to /tmp).

The postinstall script then executes the shell script in /tmp, which executes the necessary pkgrm command against the old package and then deletes itself.

This example illustrates a basic upgrade. It is less than fifty lines of code including some fairly long messages. It could be expanded to backout the upgrade or make other major transformations to the package as required by the designer.

The design of the user interface for an upgrade option must be absolutely sure that the administrator is fully aware of the process and has actively requested upgrade rather than parallel installation. There is nothing wrong with performing a well

understood complex operation like upgrade as long as the user interface makes the operation clear.

The `request` Script

```
# request script
control an upgrade installation

PATH=/usr/sadm/bin:$PATH
UPGR_SCRIPT=/tmp/upgr.$PKGINST

UPGRADE_MSG="Do you want to upgrade the installed version ?"

UPGRADE_HLP="If upgrade is desired, the existing version of the \
 package will be replaced by this version. If it is not \
 desired, this new version will be installed into a different \
 base directory and both versions will be usable."

UPGRADE_NOTICE="Conflict approval questions may be displayed. The \
 listed files are the ones that will be upgraded. Please \
 answer \"y\" to these questions if they are presented."

pkginfo -v 1.0 -q SUNWstuf.\*

if [ $? -eq 0 ]; then
   # See if upgrade is desired here
   response=`ckyorn -p "$UPGRADE_MSG" -h "$UPGRADE_HLP"`
   if [ $response = "y" ]; then
      OldPkg=`pkginfo -v 1.0 -x SUNWstuf.\* | nawk ' \
      /SUNW/{print $1} '`
      # Initiate upgrade
      echo "PATH=/usr/sadm/bin:$PATH" > $UPGR_SCRIPT
      echo "sleep 3" >> $UPGR_SCRIPT
      echo "echo Now removing old instance of $PKG" >> \
      $UPGR_SCRIPT
      if [ ${PKG_INSTALL_ROOT} ]; then
         echo "pkgrm -n -R $PKG_INSTALL_ROOT $OldPkg" >> \
         $UPGR_SCRIPT
      else
         echo "pkgrm -n $OldPkg" >> $UPGR_SCRIPT
      fi
      echo "rm $UPGR_SCRIPT" >> $UPGR_SCRIPT
      echo "exit $?" >> $UPGR_SCRIPT

      # Get the original package's base directory
      OldBD=`pkgparam $OldPkg BASEDIR`
      echo "BASEDIR=$OldBD" > $1
      puttext -l 5 "$UPGRADE_NOTICE"
   else
       if [ -f $UPGR_SCRIPT ]; then
          rm -r $UPGR_SCRIPT
```

(continued)

```
      fi
    fi
fi

exit 0
```

The `postinstall` Script

```
# postinstall
to execute a simple upgrade

PATH=/usr/sadm/bin:$PATH
UPGR_SCRIPT=/tmp/upgr.$PKGINST

if [ -f $UPGR_SCRIPT ]; then
   sh $UPGR_SCRIPT &
fi

exit 0
```

Glossary

application binary **interface** **ABI**	Definition of the binary system interface between compiled applications and the operating system on which they run. See application binary interface (ABI).
base directory	The location where relocatable objects will be installed. It is defined in the `pkginfo` file, using the `BASEDIR` parameter.
build time	The time during which a package is being built with the `pkgmk` command.
build variable	A variable that begins with a lowercase letter and is evaluated at build time.
class	A name that is used to group package objects. See also class action script.
class action script	A file that defines a set of actions to be performed on a group of package objects.
collectively **relocatable object**	A package object that is located relative to a common installation base. See also base directory.
composite package	A package that contains both relocatable and absolute path names.
`compver` **file**	A method of specifying package backward-compatibility.
control file	File that controls how, where, and if a package is to be installed. See information file and installation script.
copyright	The right to own and sell intellectual property, such as software, source code, or documentation. Ownership must be stated on the

171

CD-ROM and insert text, whether the copyright is owned by SunSoft, or by another party. Copyright ownership is also acknowledged in SunSoft documentation.

`depend` **file**	A method of resolving basic package dependencies. See also `compver` file.
incompatible package	A package that is incompatible with the named package. See also `depend` file.
individually relocatable object	A package object that is not restricted to the same directory location as a collectively relocatable object. It is defined using an install variable in the *path* field in the `prototype` file, and the installation location is determined via a `request` script or a `checkinstall` script.
information file	A file that can define package dependencies, provide a copyright message, or reserve space on a target system.
installation script	A script that enables you to provide customized installation procedures for a package.
install time	The time during which a package is being installed with the `pkgadd` command.
install variable	A variable that begins with an uppercase letter and is evaluated at install time.
package	A collection of files and directories required for a software application.
package abbreviation	A short name for a package that is defined via the `PKG` parameter in the `pkginfo` file.
package identifier	A numerical suffix added to a package abbreviation by the `pkgadd` command.
package instance	A variation of a package, which is determined by combining the definitions of the `PKG`, `ARCH`, and `VERSION` parameters in the `pkginfo` file for the package.
package object	Another name for an application file that is contained in a package to be installed on a target system.
parametric path name	A path name that includes a variable specification.

patch list	A list of patches that affect the current package. This list of patches is recorded in the installed package in the `pkginfo` file.
prerequisite package	A package that depends on the existence of another package. See also `depend` file.
procedure script	A script that defines actions that occur at a particular point during package installation and package removal.
relocatable	A package object defined in a `prototype` file with a relative path name.
relocatable object	A package object that does not need an absolute path location on a target system. Instead, its location is determined during the installation process. See also collectively relocatable object and individually relocatable object.
reverse dependency	A condition when another package depends on the existence of your package. See also `depend` file.
segmented	A package that does not fit on a single volume, such as a floppy disk.
`tar`	Tape archive retrieval. Solaris command for adding or extracting files from a media.

Index

format of, 20
how to create, 32
in a case study, 90, 94, 99, 102, 105, 107, 110, 113
using environment variables in, 12
valid file types, 21

R

relocatable object, 22
relocatable package, 134
    traditional example, 134
relocation
    supporting in a heterogeneous environment, 133
removef command, 60, 83, 146
    in a case study, 112
removing classes, 62
request script, 5, 43, 49, 50, 87, 124 to 125, 129, 146, 168, 170
    and environment variables, 51
    and package removal, 51
    behaviors, 54, 57
    design rules, 54
    example, 56, 58
    how to write a, 55
    in a case study, 91, 111
    writing a, 54
reserving additional space on a target system, 47

reverse dependency, 43

S

scripts*, see* installation scripts,
sed class
    script, 65
        in a case study, 103, 116
software package*, see* package,
space file, 4, 47
    example, 49
    how to create a, 48
    in a case study, 94
system object classes, 64

T

transferring a package to a distribution medium, 84

U

unbundled packages, 138
upgrading packages, 168

V

verifying package installation, 75
    the process, 71