

Application Packaging Developer's Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.



SunSoft
A Sun Microsystems, Inc. Business

© 1994 Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation are 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 Sun and its licensors, if any.

Portions of this product may be derived from the UNIX[®] and Berkeley 4.3 BSD systems, licensed from UNIX System Laboratories, Inc., a wholly owned subsidiary of Novell, Inc., and the University of California, respectively. Third-party font software in this product is protected by copyright and licensed from Sun's font suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the United States Government is subject to the restrictions set forth in DFARS 252.227-7013 (c)(1)(ii) and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Sun Microsystems Computer Corporation, SunSoft, the SunSoft logo, Solaris, SunOS, OpenWindows, DeskSet, ONC, ONC+, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and certain other countries. UNIX is a registered trademark of Novell, Inc., in the United States and other countries; X/Open Company, Ltd., is the exclusive licensor of such trademark. OPEN LOOK[®] is a registered trademark of Novell, Inc. PostScript and Display PostScript are trademarks of Adobe Systems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCstorage, SPARCware, SPARCcenter, SPARCclassic, SPARCcluster, SPARCdesign, SPARC811, SPARCprinter, UltraSPARC, microSPARC, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun[™] Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the Massachusetts Institute of Technology.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.



Contents

1. Distributing Software on CD-ROM	1
Introduction	1
Manufacturing a CD-ROM	2
Documentation	4
CD-ROM Packaging	5
Putting Your Software on CD-ROM	8
File System Formats	12
Installing Software from CD-ROM	15
Software Packaging	16
Using Rock Ridge to Create a CD-ROM	17
CD-ROM File System Creation Procedure	18
2. Application Packaging	21
What Are Packages?	21
Licensing Considerations for Packages	23
Installation Media	24

Package Components	26
Required Package Components	28
Optional Package Components	29
Package Objects	30
A Package Creation Scenario	31
Assigning a Package Abbreviation	33
Defining a Package Instance	33
Defining Object Classes	35
Defining Package and Object Locations	37
Writing Your Installation Scripts	39
Creating the <code>pkginfo</code> File	39
Defining Package Dependencies	40
Writing a Copyright Message	41
Reserving Additional Space on the Installation Machine ..	41
Distributing Packages over Multiple Volumes	42
Creating the <code>prototype</code> File	42
Creating a Package with <code>pkgmk</code>	49
Transferring the Package to the Media with <code>pkgtrans</code> ...	52
Installation Scripts	54
The Request Script	59
The Procedure Script	60
The Class Action Script	61
3. Installing and Checking Packages	69
The Installation Software Database	70

Installing Software Packages	71
Interacting with <code>pkgadd</code>	71
Installing Packages for Clients on a Server	72
Checking Installation Accuracy	73
Displaying Information About Installed Packages	74
4. Creating Icons and Package Clusters	79
Creating an Icon for Your Application	79
Using Clusters	79
Contents and Order Files	80
A. Packaging Guidelines	83
Optimize for Client-Server Configurations	83
Package by Functional Boundaries	83
Package Along Royalty Boundaries	84
Package by Machine Dependencies	84
Overlap in Packages	84
Sizing Considerations	84
Localization Software Packaging Guidelines	84
B. Packaging Case Studies	85
Introduction	85
Case #1: Using a <code>request</code> Script	85
Case #2: Using Classes, Class Action Scripts, and the <code>space</code> File	90
Case #3: Using <code>copyright</code> , <code>compver</code> , and <code>depend</code> Files ..	94
Case #4a: Using the <code>sed</code> Class and a <code>postinstall</code> Script	96

Case #4b: Using Classes and Class Action Scripts.....	99
Case #4c: Using the build Class.....	102
Case #5: Using crontab in a Class Action Script.....	104
Case #6: Installing a Driver.....	108
Case #7:Using the sed Class and postinstall and preremove Scripts.....	111
Glossary.....	123

Tables

Table P-1	Typographic Conventions	xiii
Table 1-1	CD-ROM Levels and File Formats	12
Table 2-1	Packaging Commands and Files	22
Table 2-2	Packages Run on Different Hardware	34
Table 2-3	Packages Run on the Same Hardware	35
Table 2-4	Installation Parameters	57
Table 2-5	Installation Script Exit Codes	58
Table 3-1	Package Parameters	75

Figures

Figure 1-1	CD-ROM Development Cycle	3
Figure 1-2	Summary of Documentation Process	7
Figure 1-3	CD-ROM Manufacturing Process	11
Figure 2-1	The Contents of a Package	26
Figure 2-2	Packaging Overview	27
Figure 4-1	CD-ROM Directory Hierarchy Example	80
Figure 4-2	Hierarchical Directory Structure	116
Figure 4-3	Flat Directory Structure	117

Preface

Purpose of This Guide

The *Developer's Guide to Application Packaging* describes how to prepare your software for distribution on CD-ROM. This guide does not cover application design, user interface design, retrieval software, or multimedia. You should have completed coding and testing your software before you begin work on package creation.

Audience

This guide is for developers who are writing applications intended to run under Solaris® system software.

Organization of this Guide

This guide is organized as follows:

Chapter 1, “Distributing Software on CD-ROM,” describes the tasks required to put your software on CD-ROM for distribution.

Chapter 2, “Application Packaging,” describes the application packaging tools.

Chapter 3, “Installing and Checking Packages,” describes the tools for verifying that a package has been installed correctly.

Chapter 4, “Creating Icons and Package Clusters,” describes how to use clusters and meta-clusters when packaging software.

Appendix A, “Packaging Guidelines,” describes the application packaging guidelines to be followed when creating software packages.

Appendix B, “Packaging Case Studies,” provides several examples of creating application packages.

For More Information

For background information on packaging, refer to:

- System V Application Binary Interface (UNIX Press)
- **SPARC systems:** System V Application Binary Interface SPARC™ Processor Supplement (UNIX Press)
- **x86 systems:** System V Application Binary Interface Intel386 Processor Supplement

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols 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 <code>.login</code> file. Use <code>ls -a</code> to list all files. system% You have mail.
AaBbCc123	What you type, contrasted with on-screen computer output	<pre>system% su Password::</pre>
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> be root to do this.
Code samples are included in boxes and may display the following:		
%	UNIX C shell prompt	system%
\$	UNIX Bourne and Korn shell prompt	system\$
#	Superuser prompt, all shells	system#

Manual Page References

When commands are mentioned in the text for the first time, a reference to the command's manual page is included in parentheses: `command(section)`. The numbered sections are located in the *Solaris 2.5 Reference Manual AnswerBook*.

Information in the Examples

The examples in this guide match what you see on the screen as closely as possible. However, your system may have a different configuration or be running a different release of the SunOS operating system.

Complete code samples should compile and work as represented. Code fragments, while not compiled, reflect high standards of coding accuracy.

Introduction

After you have completed development of your software, you need to put the software on medium in a form that can be easily installed by users. The medium should be packaged with any documentation required for your product before being distributed to customers.

CD-ROM is the best medium available for the distribution of data. For this reason, all SunSoft software is released on CD-ROM. Some features of CD-ROM include:

- *Large capacity* – 644 megabytes of digital data, or 325,000 pages, saving several trees.
- *Multimedia* – Can contain text, images, graphics, and high quality sound data.
- *Portable* – Unlike hard disks, CD-ROMs can be easily moved.
- *Stable storage* – CD-ROMs are optical, not magnetic, and are read-only. They can't be accidentally erased or overwritten.
- *Low cost* – About \$2 each to produce. Making 100 CD-ROMs is break-even with tape.
- *Mass produced* – Injection molded, not magnetically duplicated.
- *High quality* – Digital error correction means fewer data errors.
- *Interchangeable* – All CD-ROMs are the same at the bit level. Almost any CD-ROM player can read ISO 9660 formatted CD-ROMs.
- *Interactive* – Random-access file system allows execution from the CD-ROM.

CD-ROMs are random-access devices that can be directly mounted by the operating system. Unlike tapes, CD-ROMs are not limited to serving as serial input/output devices.

Software can be published on a CD-ROM in a serial format such as `tar` or can be published as a complete file system. The latter is preferred since it enables the individual data files to be directly accessed or executed from the CD-ROM.

Direct access and execution benefits the user because the software does not have to consume scarce disk resources before it is used. For trial and demonstration uses, installation time can be drastically reduced since the CD-ROM file system only has to be mounted, not copied and installed onto hard disk.

This chapter describes how to create a CD-ROM image for your application and prepare it for mass duplication. An *image* is a device-independent electronic representation of your data and software.

The SVR4 application packaging tools provide a means to bundle the files for your application into one installable unit. You provide several text files describing the contents of the package and where the contents should be installed. Chapter 2, “Application Packaging,” describes how to use the SVR4 packaging tools to create a package that can be placed on CD-ROM.

After you have created a package, you should verify that it installs correctly. Tools such as `pkgchk(1M)` help you verify the installation. Chapter 3, “Installing and Checking Packages,” describes these tools.

You may want to use `admintool` for installation of your application. The `admintool` software provides a graphical user interface to the SVR4 packaging tools.

Manufacturing a CD-ROM

Several tasks are required to produce a CD-ROM and bring it to publication. Experience at SunSoft shows that some of these tasks can be done at the same time, in particular, documentation and software development. Figure 1-1 illustrates the development cycles for each. If you can perform these processes independently, you can save time.

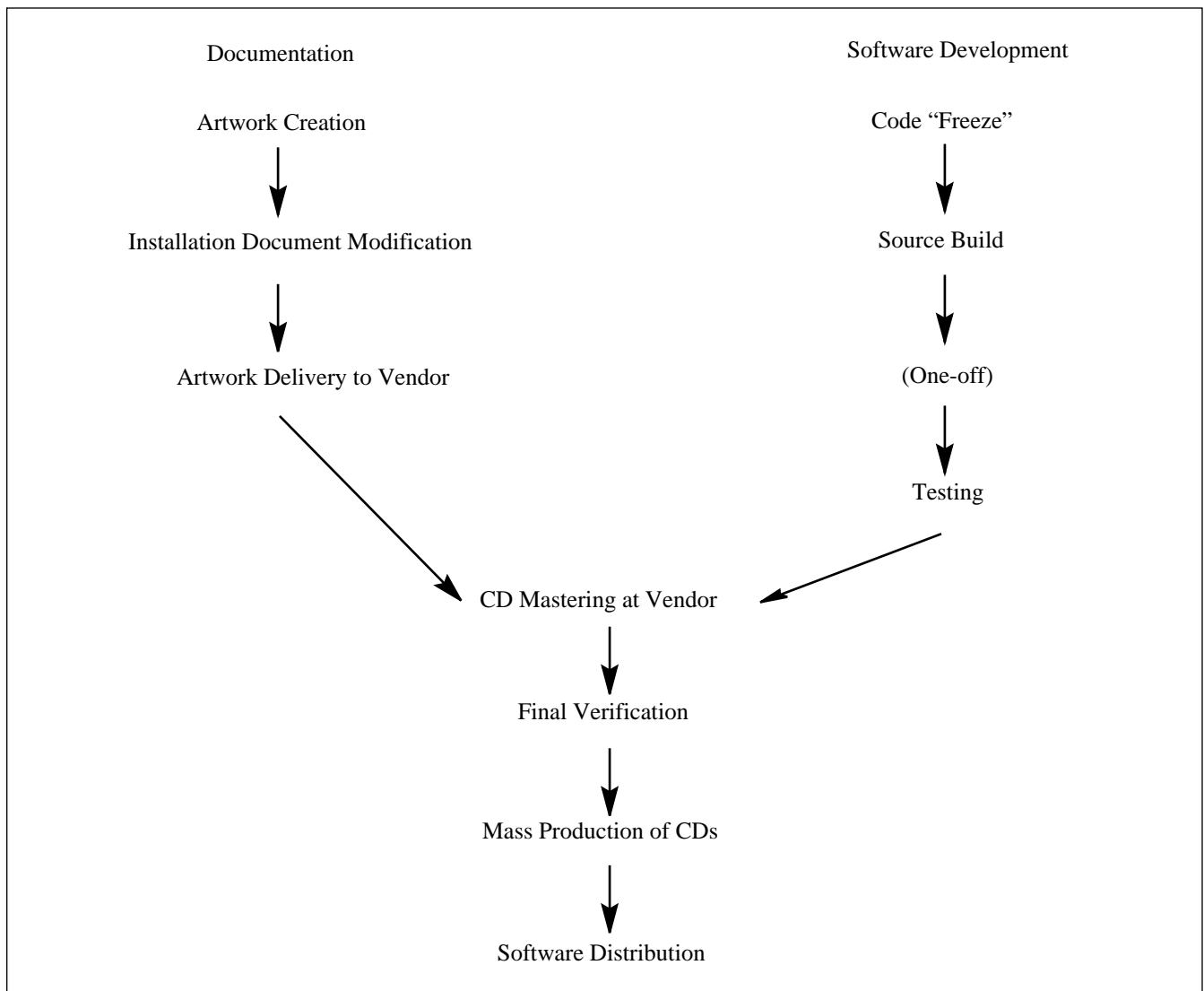


Figure 1-1 CD-ROM Development Cycle

Documentation

Three kinds of documentation can accompany a CD-ROM:

- Documentation and artwork printed on heavy insert stock
- Artwork silkscreened on the disc itself
- Separate, traditional documentation printed on paper

The traditional paper documentation for CD-ROM will probably differ only slightly from your documentation for tape release. You will need to modify your installation instructions to discuss CD-ROM installation and may want to add a short discussion of the CD-ROM medium.

You also need to consider what type of paper documentation is needed. Typically, there are three kinds of paper documentation: user's guide, developer's guides, and system administration manuals. Your application may only require a user's guide. But if a system administrator needs to install and configure the software, you probably want to include a manual or section for administrators, too.

If manual pages accompany your software, you need to make sure the manual pages have the correct section numbers. The contents of the manual page sections are as follows:

- Section 1 - describes user commands and applications
- Section 1M - describes system administration commands
- Section 2 - describes system calls
- Section 3 - describes user-level library routines
- Section 4 - describes device drivers, protocols, and network interfaces
- Section 5 - describes the format of files used by various programs
- Section 6 - describes games
- Section 7 - contains miscellaneous information, mostly relating to `troff` macro packages
- Section 9 - provides an overview of DDI/DKI device driver interface specifications
- Section 9E - describes DDI/DKI driver entry points
- Section 9F - describes DDI/DKI kernel functions
- Section 9S - describes DDI/DKI data structures

CD-ROM Packaging

Each CD-ROM can be packaged in a *jewel box*, a small plastic case, or a less expensive paper sleeve. In addition to the CD-ROM disc, the jewel box contains an *insert* and a *J-card*.

Insert

The insert slides into the front cover of the CD-ROM jewel box, and usually serves as the product label. Inserts usually include the file system format type, part numbers, and trademark and copyright information associated with the specific product. If your documentation is short enough, you can include it in the insert, as well.

The insert can be one page with text and artwork, or it can be several pages long. In fact, a small booklet can be produced, with artwork for the cover of the jewel box and text describing the product and giving simple installation instructions. Keep in mind, however, that the printing may be quite small and difficult to read on the insert.

J-card

The optional J-card is a printed card with a small folded edge that fits into the back of the CD-ROM jewel box. It typically has the product name and part number, which can be read by the user without opening the jewel box. It serves the same function as printing on the spine of a book. The J-card can give basic information instead of an insert, allowing the artwork on the disc itself to show through the front cover.

The artwork for the J-card and CD-ROM insert are prepared by a graphic artist. The artwork is assembled at the manufacturer's location with the CD-ROM and jewel box.

CD-ROM Artwork

The CD-ROM itself can have label art, but the amount of information that can be put on the label is limited because of the CD-ROM size. The disc artwork should include product name, company name and address, part number, revision level, trademark, copyright, and a list of platforms on which it runs. It can also have limited boot instructions or state the file system format type.

Delivery to the CD-ROM Manufacturer

Documentation production has a long lead time. Because the printing is done by an external vendor for your company, you have limited control of the schedule. The CD-ROM manufacturer, or its subcontractor, usually takes four weeks to produce a first article (a completed CD-ROM used to verify artwork and contents) from camera ready artwork. This time could be accelerated by coordinating your own printing locally, but your printed materials would have to be integrated with the finished CD-ROM after the manufacturing process. Schedules for lead times can be negotiated with your manufacturer.

Contact the CD-ROM manufacturing company early to discuss requirements and limitations for creating and printing insert text and graphics, because each vendor has different specifications.

Plan to deliver the camera ready disc artwork, the insert copy artwork, and the optional J-card artwork to the CD-ROM manufacturer. The manufacturer places the printed material and the disc in the jewel box and then shrink-wraps the jewel box in clear plastic.

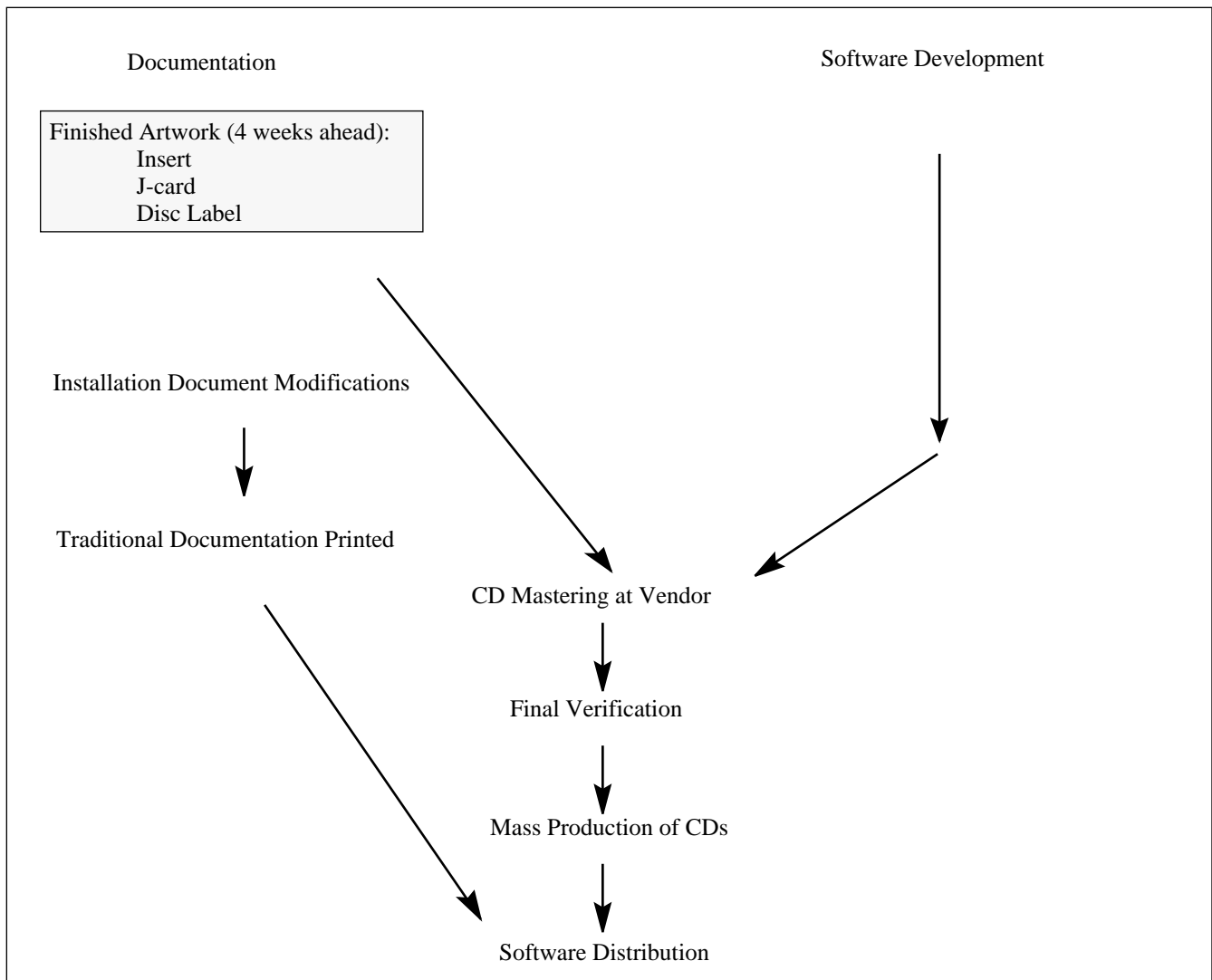


Figure 1-2 Summary of Documentation Process

Putting Your Software on CD-ROM

The simplest way to make a CD-ROM for distribution is to create a CD-ROM image of your application on magnetic disk and transfer that image to tape for delivery to a CD-ROM mass producer. However, this approach gives you no opportunity to test an actual CD-ROM version of your product. You may also want more control or require more involvement in the process.

Every company has a different process to release and manufacture a product. At a minimum, the following steps are necessary to transfer your software to CD-ROM:

1. Completing the application code.
2. Building the source into an executable media image.
3. Testing the image and installation.
4. Verifying the master image.
5. Mass producing the CD-ROM media (by the manufacturer).
6. Receiving the finished goods and combining with documentation for distribution.

The rest of this section explains each of these steps in more detail.

1. After the application code source is complete, all the data that makes up the product is collected in one location. The file system structure layout must be determined by this time. See the section “File System Formats” on page 12.
2. At build time, quality audits are done and a prototype executable is made. When the prototype media image is satisfactorily built, you have to choose how to hand off this image to be tested. Because CD-ROM is a read-only random-access file system, this image could be simulated electronically, using a read-only magnetic disk image of the CD-ROM file system for testing on a magnetic disk partition. If applicable to your product, this *electronic handoff* would provide you more opportunities for testing before you actually create a CD-ROM.

A more complex option is to use an actual CD-ROM for testing. The CD-ROM can be created in a variety of ways. One common way is to take a magnetic tape with a CD-ROM image to a CD-ROM service bureau. For a

fee, usually \$600 to \$800 (check with various service bureaus), the service bureau creates a CD-ROM using a *pre-mastering* or *one-off* machine. Many CD-ROM manufacturing facilities offer these services.

You could also purchase your own one-off system, if your CD-ROM production quantity requires it. A one-off machine produces each copy of the CD-ROM individually, so a one-off machine is not usually used for high volume manufacturing of CD-ROMs. The CD-ROM medium itself is more expensive than tape and can cost from \$35 to \$80 each, depending on volume and discount.

3. The media image is handed off to testing. The image must be tested against all configurations of systems on which it is expected to operate. It is extremely important to test the installation process. For manual or automated installation testing, having a CD-ROM one-off is invaluable.
4. When all tests have been completed, the final image must be captured for mass production. When you want to *cut* a CD-ROM, you need to put it on the medium that your CD-ROM manufacturing facility can accept. Contact the manufacturer for the appropriate medium. In many cases the manufacturer expects the master image on a tape, sometimes on EXATAPE® 8mm data cartridges.

An outline of the basic steps to lay out your file system for the Rock Ridge format (the recommended file system format for CD-ROMs) is provided in “Using Rock Ridge to Create a CD-ROM” on page 17. The software toolkit for creating a CD-ROM in the Rock Ridge format is available from Young Minds, Inc. Please consult your Catalyst Catalog.

If your application contains any audio data, you will need to contact the manufacturer to find out the medium and format required to handle this data. For example, some manufacturers require that you provide this data on Digital Audio Tape (DAT) at 44.1 KHz.

5. If you use an external CD-ROM manufacturer, you can expect a turnaround ranging from one day to several days, depending on the quantity and how much you are willing to pay for faster turnaround. In general, it costs about \$800 to \$2500 to set up the master, and approximately \$2 for each CD-ROM produced. Required turnaround time and quantities affect these numbers. The process at the CD-ROM manufacturing factory usually follows the path shown in Figure 1-3.

The CD-ROM manufacturer transfers the image from your tape master to a glass master. From this, a metal stamping tool is produced. This tool is used to manufacture the CD-ROMs. A limited number of CD-ROMs, called *check discs* or *first articles*, can be produced and returned to you for a final media check against your original media image.

Depending on the manufacturer, the media check can be done at several different stages. Some manufacturers may produce a *data proof* for early review. A data proof is a disc with a generic label but the actual data image on the disc. For review at a later stage a manufacturer may produce a first article, a disc with the actual data image and the actual label artwork.

The media check is an optional step that can take time, but it is your safety net. If there is any discrepancy, you may save yourself the cost of remastering and redoing an expensive mass production run. The cost and conditions for these review services can be negotiated with your CD-ROM manufacturer.

When all is satisfactory, the CD-ROM manufacturer can begin mass production of your CD-ROMs.

6. The CD-ROM manufacturer returns CD-ROMs to you in jewel boxes with optional inserts, all in shrink-wrapped packages. These can then be packaged with documentation for distribution. For an additional fee, some manufacturers handle the distribution as well. Check with them for fees and schedules.

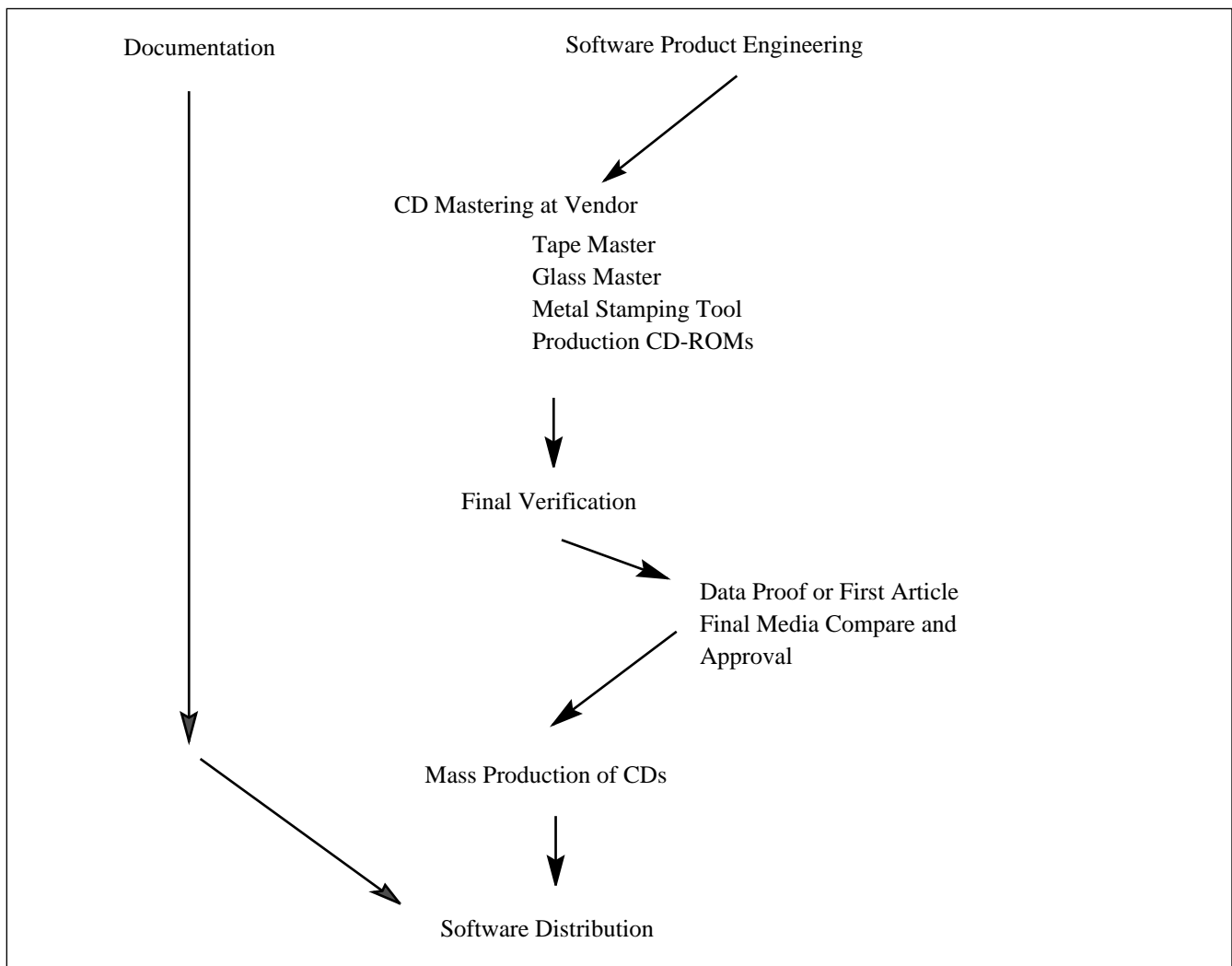


Figure 1-3 CD-ROM Manufacturing Process

File System Formats

There are nine levels of information on a CD-ROM as shown in Table 1-1. There are standards for the lowest three levels of the CD-ROM (file/volume, data, and physical). This section discusses the file system format (file/volume in Table 1-1) level.

Table 1-1 CD-ROM Levels and File Formats

User Interface
Applications
Operating System Extensions
Device Driver
Hardware Interface
Drive/Commands
File/Volume (High Sierra, ISO 9660, UFS, Rock Ridge)
Data
Physical

High Sierra

In 1985, several CD-ROM and computer manufacturers met and agreed upon a common format for file systems on CD-ROM. The format covers the logical structure (file system format); the physical structure is the same as for music CD-ROMs. This logical structure became known as the High Sierra File System (HSFS).

HSFS fit naturally into a DOS environment, supporting various DOS features and naming conventions. Unfortunately, it did not support several UNIX® features.

ISO 9660

In 1988, the International Organization for Standardization (ISO) adopted a superset of the HSFS requirements as the ISO 9660 standard. This standard included support for the VMS® operating system.

Though any file system format can be used, the international standard for CD-ROM is ISO 9660. Many different types and classes of computers read ISO 9660, allowing data interchange among the different ISO 9660 compatible systems. For example, a database application on a CD-ROM could contain the data, plus access software for UNIX, Apple Macintosh®, MS-DOS® and other types of systems.

Unfortunately, ISO 9660 is a least-common-denominator approach and has several limitations in an IEEE/POSIX or X/Open environment such as the SunOS operating system:

- File names are limited in length and allowable characters
- The depth of subdirectories is limited to seven
- File mode bits are not fully supported

UFS

Since any bit pattern can be written to a CD-ROM, alternatives to ISO 9660 can be used. For example, the file format used by Solaris on hard disks, UFS, can also be used on CD-ROM. However, UFS is not effective for CD-ROM use because UFS was designed for both reading and writing. Therefore, UFS can be slower than ISO 9660. More importantly, UFS is not a CD-ROM standard and does not offer the data interchange capabilities of ISO 9660. The advantage of UFS is that it does not have the naming and other limitations of ISO 9660.

Rock Ridge Extensions to ISO 9660

To provide UNIX functionality on the ISO standard, a set of UNIX extensions has been adopted as a solution. These extensions are known as the Rock Ridge Extensions. Rock Ridge adds all the functionality UNIX needs in the file system, such as directory depths greater than eight levels and symbolic links to files.

Features of the Rock Ridge Extensions

The Rock Ridge extensions support capabilities that are not available under the MS-DOS or VAX VMS operating systems for which the ISO 9660 was designed. These capabilities include support for mixed-case names, long filenames, special characters, directory structures deeper than seven levels, symbolic links, special file types, setuid, setgid and sticky bits, as well as more efficient encoding of user and group IDs and permissions.

When mounted on a system that supports the Rock Ridge format, such as Solaris 1.x system or later, all the UNIX file system information is available to the user. On systems that support only the ISO 9660, all the file content is available to the user (minus some of the UNIX file system information), supporting the major goal of the ISO 9660 information interchange.

The combination of the ISO 9660 and the Rock Ridge extensions provide an exceptional blend of flexibility and performance. Rock Ridge CD-ROMs execute 25% to 30% faster than UFS images recorded on CD-ROMs, yet support complete POSIX file system semantics. Further, the Rock Ridge protocols are a nonproprietary, open specification being implemented by many of the major UNIX vendors. The protocols provide support within heterogeneous networks, including multiple product lines from a single vendor. Multiplatform software vendors can also use a single Rock Ridge format CD-ROM to distribute their products for many or all the platforms they support.

Choosing a File System Format

As a software developer who wants to publish software via CD-ROM, you have four options:

- ISO 9660 format with `tar` (or `cpio` or similar) files. The tar files can contain long names and unlimited subdirectories. The files cannot be directly executed or read from the CD-ROM. Existing Solaris applications can easily be ported with this method. The CD-ROM can be read by most systems, allowing software for multiple systems to be published on the same CD-ROM.
- ISO 9660 format as a file system. The files must conform to ISO 9660 naming restrictions. For preexisting UNIX applications, this requirement can create a great deal of work. SunSoft does not provide third-party tools for producing CD-ROMs using the ISO 9660 file system format.
- UFS format as a file system. The files can be directly read and executed from the CD-ROM. There are no additional file naming restrictions beyond those in Solaris. Existing Solaris applications can be ported to CD-ROM easily; however, performance is not optimal because this is not the CD-ROM standard.

- Rock Ridge format as a file system. When the CD-ROM is created, a utility can be used to automatically create ISO 9660 file names from the longer UNIX file names. “Makedisc” is a utility with this capability; it is available from Young Minds, Inc., a Catalyst vendor: (714) 335-1350. Check your Catalyst Catalog for others.

If you are interested in using the Rock Ridge file system format for your CD-ROM image, refer to the outline of basic steps provided in the section “Using Rock Ridge to Create a CD-ROM” on page 17.

The Solaris system software supports CD-ROMs encoded using High Sierra, ISO 9660, UFS, and Rock Ridge formats.

Installing Software from CD-ROM

The goal of installation is to move code from a distribution medium to a customer’s system. CD-ROM simplifies this, because the customer handles only one CD-ROM instead of multiple tapes or diskettes.

Your installation instructions should tell the user to insert the CD-ROM into the CD-ROM *caddy* before inserting it into the CD-ROM drive. Some CD-ROM devices, like home audio units, do not have a caddy.

The user does not need to mount the CD-ROM; it is automatically mounted by the volume management software.

The installation method you choose depends upon the product format you select for laying out your files on the CD-ROM. The product format is largely independent of the file system you choose, with the exception of the limitations already mentioned for ISO 9660.

For Solaris 2.x releases, Sun is standardizing on the Rock Ridge file system format for both OS and unbundled product CD-ROMs. SVR4 *software packages* is the standard API that is used for the product format, both for OS and unbundled product CD-ROMs. The Solaris `admintool` can be used as an easy-to-use installation tool and as a frontend to the Package utilities such as `pkgadd` and `pkgrm`. The `admintool` is bundled with Solaris 2.x and is available for use as an installation tool for any product that uses SVR4 Packages. See the *Solaris Advanced User’s Guide* and the `admintool(1M)` man page for more information.

A CD-ROM file system toolkit is a set of tools and utilities that enables you to easily transition to CD-ROM for distributing computer-based materials. The critical component is a CD-ROM formatting utility that converts a UNIX file system to a CD-ROM disc image compliant with the ISO 9660 international standard format. If the utility also supports the Rock Ridge extensions to the ISO 9660, the resulting CD-ROM retains all the UNIX file system features.

Software Packaging

The System V ABI specifies a new model, called software packages, for the distribution format of applications. Software that is formatted with the ABI model is guaranteed to install correctly, easily, and in a similar manner on all ABI-compliant systems. All software producers, including applications programmers and developers of device drivers, kernel modules, and other system software for Solaris 2.x, should use the software packages model and packaging tools. See Chapter 2, “Application Packaging,” for more information on the packaging tools.

If you distribute your software as one or more software packages, you can instruct the user to install this software with either of the package installation facilities bundled with Solaris: generic ABI package commands (`pkgadd`) or the `admintool`.

Generic Package Interface

If you document the ABI package commands as your preferred means of installation, you need to provide instructions for each of the following functions:

- Invoking `pkgadd`

See Chapter 2, “Application Packaging,” for more information on using `pkgadd`.

- Installing the software on diskless and dataless clients

See Chapter 3, “Installing and Checking Packages,” for information on installing software on clients.

You do not need to provide instructions for mounting the CD-ROM. The CD-ROM is mounted automatically by the volume management software when it is inserted into the drive. If volume management isn't running, `pkgadd` mounts the volume.

Executing Applications from CD-ROM

CD-ROM can be used for execution as well as distribution. CD-ROM is a random-access file system, so you can execute your application directly from the mounted CD-ROM without installing it onto a magnetic disk first. This can save magnetic disk resources.

There are three things that must be considered when executing directly from CD-ROM:

- Make sure the application does not try to create files on the distribution file system. For example, don't let the application write a log file to `./logfile`. In the past, this was not a problem because you loaded the product onto magnetic disk where you had write permission. But if you execute directly from the CD-ROM, you cannot write to it, because it is a read-only medium.
- The application should not rely on an absolute mount point. The application should use path names that are relative to the mount point instead.
- Performance. An optical disk is slower than a magnetic hard disk.

Using Rock Ridge to Create a CD-ROM

This section describes how to put application software on a CD-ROM that is usable under Solaris system software. The procedures described in this section explain how to make a mountable UFS file system, containing your software, that can be transferred to CD-ROM. The topics covered include:

- How to create a file system you can use on CD-ROM
- Transferring your files to this file system
- Making an image of the file system that can be used to create a CD-ROM

At the end of this section is a brief discussion of the steps required to make a mountable Rock Ridge file system.

This material does not describe how to prepare audio tracks to be placed on CD-ROM, nor does it address the specific techniques for transferring the file system or audio tracks to a CD-ROM. These techniques should be explained by the CD-ROM mastering machine documentation.

CD-ROM File System Creation Procedure

The steps for creating a file system to put on a CD-ROM are as follows:

1. Determining which files will be on the CD-ROM
2. Finding a disk partition for the file system
3. Creating the file system
4. Mounting the file system
5. Transferring your files to the file system
6. Unmounting the file system
7. Making an image of the file system
8. Testing the image before transferring it to CD-ROM

Determining Which Files Go on the CD-ROM

Choose the directories and files that you want to put on the CD-ROM file system. Keep in mind that this is a read-only file system.

You should create a clean directory structure containing only the files and directories you want to put on the file system. This makes the tasks of determining the size required for the file system and transferring the files to the file system much easier.

Finding a Disk Partition

Find or create a disk partition large enough to hold the file system. A reasonable minimum size is one and a half times the size of the directory hierarchy you want to place on the CD-ROM. The maximum size is the size of the CD-ROM, roughly 600 Mbytes. Choosing a size closer to the minimum saves storage space and CD-ROM creation time. It also leaves space on the CD-ROM for audio tracks.

Creating and Mounting the File System

Create a file system in the partition and mount the file system. See *System Administration Guide, Volume I* for information about creating and mounting file systems.

Transferring Your Directory Structure to the File System

Copy your files to the new file system. The directory structure should be as you want it on your CD-ROM. Make sure all file attributes, such as permissions, modes, and links, are set correctly.

Unmounting the File System

See the *System Administration Guide, Volume I* for information on unmounting file systems.

Making a Rock Ridge Image of the File System

Use the third party tool you have selected to create a Rock Ridge image of the UFS file system created in the preceding step. After the Rock Ridge file system image is created, it can be copied to a standard disk, and tested.

To copy the partition to a regular file, use `dd` as follows:

```
# dd if=/dev/rpart of=cdromimage
```

where *part* is the device for the disk partition containing the file system and *cdromimage* is the name of the file where the image should be put.

If you are copying the image to a tape, you probably have to specify block size. Block size is determined by your tape drive and by the equipment the tape will be read on. Your CD-ROM mastering equipment or the company mastering your CD-ROM will specify this. Find a suitable block size (such as 8Kbyte) and use `dd` as follows:

```
# dd if=/dev/rpart of=/dev/rmt0 obs=8k
```

where *part* is the device for the disk partition containing the file system. If needed, replace `rmt0` with the correct name for the tape drive.

If the entire image won't fit on a single tape, use a series of `dd` commands to transfer the image, as follows:

```
(insert first tape)
# dd if=/dev/rpart of=/dev/rmt0 bs=8k count=16000 skip=0
(when done, insert second tape)
# dd if=/dev/rpart of=/dev/rmt0 bs=8k count=16000 skip=16000
(then the third tape)
# dd if=/dev/rpart of=/dev/rmt0 bs=8k count=16000 skip=32000
(etc)
```

using the device for your disk partition for *part*. Replace `rmt0` with the correct name for the tape drive. Replace the 16000 following `count` with the capacity of your tape drive (in 8Kbyte blocks), and increase the `skip` value by this amount for each successive tape.

Testing Your CD-ROM Master

When you receive your CD-ROM master, check the contents. The CD-ROM is automatically mounted by the volume management software when it is inserted in the drive. The mount point will be `/cdrom/cdrom_name`.

Make sure its contents are the same as your original directory structure.

Application Packaging



This chapter describes application packages and the tools used to create a package. The following topics are covered:

- What a package is and what it is used for
- Licensing considerations for packages
- Installation media format and file organization
- Package components
- A package build scenario
- Custom installation scripts

What Are Packages?

Application software is delivered in units called *packages*. A package is a collection of files and directories required for the software product.

It is recommended that you package your application software using `pkgmk`. See “A Package Creation Scenario” on page 31 for more information on creating packages using `pkgmk`. See the System V Application Binary Interface for more information on application packages.

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

Table 2-1 shows the commands and control files available to help you create a package. These commands and files are described in more detail in the following sections and in the man pages.

Table 2-1 Packaging Commands and Files

Command or File	Purpose
pkgmk(1)	Create an installable package
pkgparam(1)	Display package parameter values
pkgproto(1)	Generate a <code>prototype</code> file for input to <code>pkgmk</code>
pkginfo(1)	Display software package information
pkgtrans(1)	Transfer and/or translate packages
installf(1M)	Add a file to the software installation database
pkgadd(1M)	Install software package onto a host
pkgask(1M)	Store answers to a request script
pkgchk(1M)	Check accuracy of an installed software package
pkgrm(1M)	Remove a package from a host
removef(1M)	Remove a file from the software installation database
admintool(1M)	Manage local systems, including adding and removing software on local system.
admin(4)	Package installation defaults file
compver(4)	Package compatible versions file
copyright(4)	Package copyright information file
depend(4)	Software dependencies file
pkginfo(4)	Package characteristics file
pkgmap(4)	Package contents description file
prototype(4)	Package information file
space(4)	Package disk space requirements file
preinstall	
postinstall	

Table 2-1 Packaging Commands and Files

Command or File	Purpose
<code>classaction</code>	
<code>preremove</code>	
<code>postremove</code>	
<code>request</code>	
<code>checkinstall</code>	

Licensing Considerations for Packages

If you are distributing software that uses licensing, there are several things you need to consider:

- Business operations
- Communication with users
- Technology

Business Operations

Before you begin distributing licensed software, set up your business operations to distribute, price, and track licenses. There are a variety of ways to distribute licenses, such as fax, electronic mail, or an 800 telephone number. You need to choose a method of distribution and set up all the necessary processes. You also need to consider whether licenses need to be upgraded with the software and how this will be done.

Pricing policy and types of licenses must also be considered. You must consider how the product is used and what kinds of licenses your users will need to use the product effectively. Single user licenses may not be appropriate for many situations.

Communication with Users

Before you implement licensing, you need to inform your users, particularly if the product has not been licensed in the past.

When you do implement licensing, you may want to consider implementing it gradually. The first step would be monitoring the use of licenses, followed by warning that the software is being used without a license, and finally, denying the use of the software.

Technology

If you are going to use a commercial product for licensing, there are many things to consider when making your choice. You need to decide what your priorities are. For example, is ease of administration and use most important? Or is enforcing the licensing policy more important?

You also need to consider whether the software will be used in a heterogeneous or homogeneous environment and whether standards are important. You may also want to look at the security provided by the product. Is it easy to get around the enforcement of licenses?

The issues involved in choosing a commercial product will vary depending on the kind of application and your reasons for implementing licensing.

Installation Media

Application software packages are installed from the distribution medium. In addition to CD-ROM distribution media, SunOS 5.x supports the physical distribution media listed in the following:

- **SPARC systems:** System V Application Binary Interface SPARC Processor Supplement
- **x86 systems:** System V Application Binary Interface Intel386 Processor Supplement

Packages are stored in data stream or directory format on the distribution media. There are two types of data streams: continuous and segmented. The continuous data stream is valid for all media. The segmented data stream is valid for media that support multiple sequential files, media with a no rewind mode of operation (such as a 9-track tape). For the segmented data stream, each of the logical parts of the data stream is an individual file. For the continuous data stream, all logical parts on a given volume of media are contained in a single file. Both data stream types can be created using the `pkgmk` and `pkgtrans` utilities.

A segmented data stream format is used whenever multiple volumes are required to contain the package. Use of this format is common when diskettes are the distribution medium, uncommon for tapes or CD-ROM. Continuous stream format is most common for tape and directory format for CD-ROM.

Packages can also be stored as a standard file system that allows distribution of multiple packages on large removable media, such as Winchester disks or optical discs. These types of distribution media are not supported by the ABI.

In a standard file system media format, the application package(s) form a tree of packages. The file system can be mounted and packages installed using `pkgadd` from that spooled package file system.

Package Components

To package your applications, you must create the required and optional components that make up your package, then use the packaging tools to build the package.

A software package has the following parts:

- Deliverable object files
- Two required metafiles
- Optional scripts and metafiles

As shown in Figure 2-1, the contents of a package fall into three categories:

- Required components (the `pkginfo` (4) file, the `prototype` (4) file, and package objects)
- Optional package metafiles
- Optional control scripts

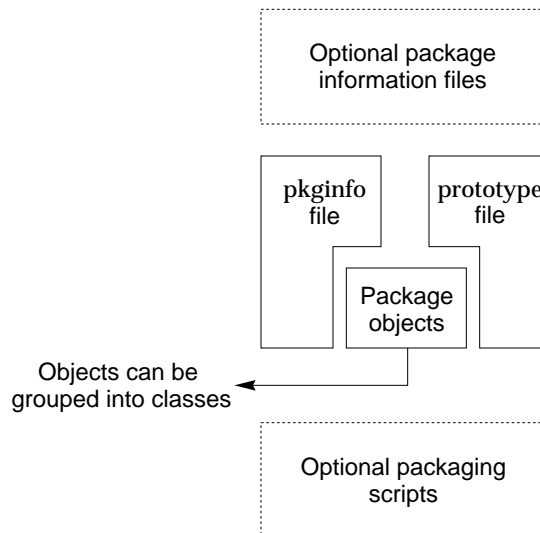


Figure 2-1 The Contents of a Package

Figure 2-2 shows an overview of the package creation process.

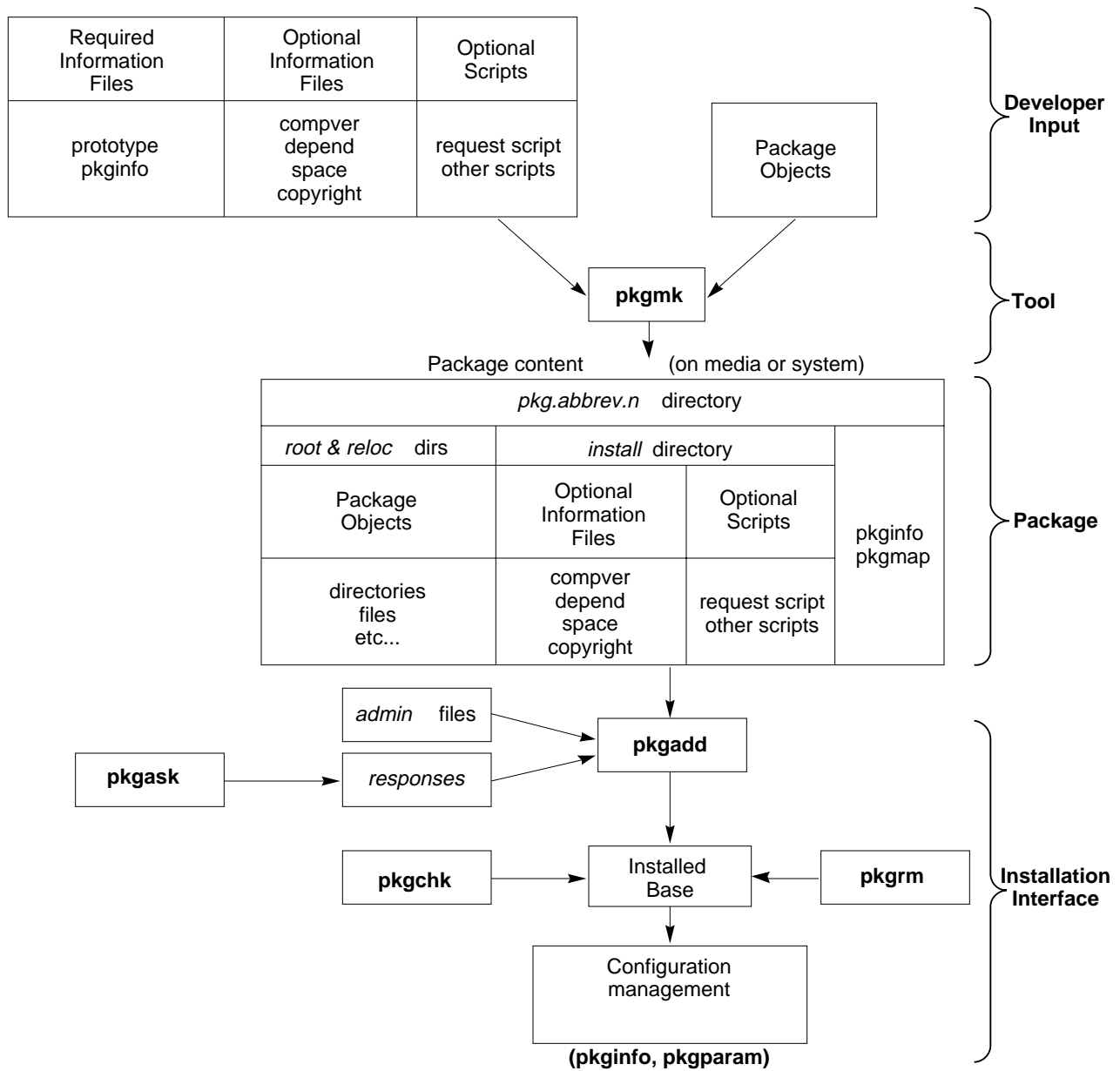


Figure 2-2 Packaging Overview

Required Package Components

A package *must* contain the following components:

- Package Objects

These are the components that make up the software. They can be

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

Objects can be manipulated in groups during installation by putting them in classes. See the section “Package Objects” on page 30” for more information on objects.

- 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. See the `pkginfo(4)` manual page for more information.

- 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.

There is one entry for each deliverable object. An entry consists of several fields of information describing the object. All package components, including the metafiles and control scripts, must be listed in the `prototype` file. See the `prototype(4)` manual page for more information.

- The `pkgmap` file

The `pkgmk` command creates the `pkgmap` file when it processes the `prototype` file. This new file contains all the information in the `prototype` file plus additional fields for each entry used for validation at install time. See the `pkgmk(1)` and `pkgmap(4)` manual pages for more information.

Optional Package Components

Package Information Files

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

- The `compver` (4) File
Defines previous versions of the package that are compatible with this version.
- The `depend` (4) File
Defines any software dependencies associated with this package.
- The `space` (4) File
Defines disk space requirements for the target environment. This is space that is required in addition to the space used by objects defined in the `prototype` file. For example, additional space might be needed for files that are dynamically created at installation time.
- The `copyright` (4) File
Defines the text for a copyright message that is displayed at the time of package installation.

Each package information file used should have an entry in the `prototype` file. All of these files are described further in the manual pages.

Installation Scripts

Installation scripts are not required. However, you can deliver scripts that perform customized actions. An installation script is comprised of Bourne commands text. It should be mode 0644 and doesn't need to contain the shell identifier (`#!/bin/sh`). The four script types are as follows:

- *request* script (requests input from the installer)
- *checkinstall* script (performs special file system tests)
- *class action* scripts (define a set of actions to perform on a group of objects)

- *procedure* script (defines actions that occur at particular points during installation)

Refer to “Installation Scripts” on page 54, for a more detailed discussion of installation scripts. Refer to Appendix B, “Packaging Case Studies,” for examples of installation scripts.

Package Objects

These are the files that are being distributed, the files that make up the application. For example, for a driver the package objects would be the loadable driver module, the hardware configuration file, the driver’s header file, and a test program.

Object Locations

You specify package object pathnames in the `prototype` file. Note that during packaging and installation, a package object can reside in any of three locations. You need to be aware of which of the three locations is being discussed. The locations are:

- Development machine

Packages originate on a development machine. They can be in a different directory structure from the installation machine. `pkgmk(1)` can locate components on the development machine and give them different pathnames on the installation machine.

- Installation media

When `pkgmk` copies the package components from the development machine to the installation medium, it places them in the locations defined in your `prototype(4)` file and in a format that `pkgadd(1M)` recognizes.

- Installation machine

`pkgadd` copies a package from the installation medium to the installation machine and puts it in the structure defined in your `pkgmap` file.

Note – Those objects that do not require an absolute path name should be specified as “relocatable.”

A Package Creation Scenario

The following is an overview of some of the steps you might use in a packaging scenario. Not all of these steps are required, and there is no mandated order for their execution.

Note – This list and the following procedures are intended as guidelines and should not replace either your own planning or reading the rest of this manual to learn about available package options. Each of the steps is explained in more detail in the following sections.

1. Assign a package abbreviation.

Every package installed in the environment must have a package abbreviation.

2. Define and identify a package instance.

Decide on values for the three package parameters that make each package instance unique. See the section “Defining a Package Instance” on page 33.

3. Define your object classes.

Decide on the classes you are going to use before you create the `prototype` file and before you write your class action scripts.

4. Define the location of the package and its objects.

Package objects can be delivered with

- Relocatable locations (they have no absolute path location requirements)
- Fixed locations (their location is defined by the package and cannot be changed)

All of a package or parts of a package can be defined as relocatable. Decide if package objects will have fixed locations or be relocatable before you write any installation scripts and before you create the `prototype` file.

Note – Use relocatable objects whenever possible. In general the major part of a package is relocatable with a few files (such as those in `/etc` or `/var`) specified as absolute.

5. Write installation scripts for your package, if needed.

Assess the needs of your package beyond the actions provided by `pkgadd` and decide which types of installation scripts are necessary to install your software.

6. Create the `pkginfo` file.

Create a `pkginfo` file before executing `pkgmk`. The `pkginfo` file should define basic information about the package and can be created with any editor as long as it follows the format described in the `pkginfo(4)` manual page. See the section “Creating the `pkginfo` File” on page 39 for more information.

7. Define package dependencies.

Determine whether your package has dependencies on other packages and if any other packages depend on yours. If so, create the `depend` file. If there are special dependencies upon files or file behaviors put that test into the `request` file or `checkinstall` for 2.5 and above.

8. Write a copyright message.

Decide whether your package must display a copyright message while it is being installed and removed. If so, create the `copyright` file.

Note – You should include a `copyright` file to provide legal protection for your application.

9. Reserve additional space on the installation machine.

Determine whether your package needs additional disk space in addition to the space required by the package objects. If so, create the `space` package information file.

10. Distribute packages over multiple volumes.

`pkgmk` automatically distributes packages over multiple volumes if the selected medium dictates that. Decide if you want to leave those calculations up to `pkgmk` or customize package placement on multiple volumes in the `prototype` file.

11. Create the `prototype` file.

This file is required and must be created before you execute `pkgmk`. It lists all the objects that belong to a package and information about each object (such as its file type and class). Create it with any editor, following the format described in the `prototype` entry in the manual pages. You can also use the `pkgproto(1M)` command.

12. Create the package.

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

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

13. Optionally, transfer the package to tape or datastream media with `pkgtrans(1M)`.

The remainder of this chapter gives procedural information for each step.

Assigning a Package Abbreviation

Each installed package must have a package abbreviation assigned to it. This abbreviation is defined with the `PKG` parameter in the `pkginfo` file.

A valid package abbreviation must meet the criteria defined below:

- It must start with a letter. Additional characters may be alphanumeric and can be the two special characters `+` and `-`.
- It must be nine or fewer characters.
- Reserved names are `install`, `new`, and `all`.

Defining a Package Instance

The same software package can have different versions, architectures or both. Multiple variations of the same package can reside on the same machine. Each variation is known as a package instance.

`pkgadd` assigns a package identifier to each package instance at installation. The package identifier is the package abbreviation with a numerical suffix. This identifier distinguishes an instance from any other package, including other instances of the same package.

Identifying a Package Instance

Three parameters defined in the `pkginfo` file combine to uniquely identify each instance. The combination of these parameters should be unique for each instance. These parameters are:

- **PKG**
Defines the software package abbreviation and remains constant for every instance of a package.
- **VERSION**
Defines the software package version.
- **ARCH**
Defines the software package architecture.

For example, you might identify two identical versions of a package that run on different hardware as shown in Table 2-2.

Table 2-2 Packages Run on Different Hardware

Instance #1	Instance #2
<code>PKG=myappl</code>	<code>PKG=myappl</code>
<code>VERSION=1.0</code>	<code>VERSION=1.0</code>
<code>ARCH=sparc</code>	<code>ARCH=intel</code>

Two different versions of a package that run on the same hardware might be identified as shown in Table 2-3.

Table 2-3 Packages Run on the Same Hardware

Instance #1	Instance #2
PKG=myappl	PKG=myappl
VERSION=1.0	VERSION=2.0
ARCH=sparc	ARCH=sparc

At the time of installation, `pkgadd` assigns a numerical suffix to the package abbreviation. The combination, for example `mypkg.2`, is known as the *package identifier*. This ID maps the three pieces of information that identify a package instance to one name, which becomes the name of this instance on your machine.

The first instance of a package installed on a system does not have a suffix, so its instance identifier will be the package abbreviation. `pkgadd` assigns subsequent instances a suffix, beginning with `.2`. An instance is given the lowest integer extension available so it may not correspond to the order in which a package was installed. For example, if `mypkg.2` was deleted after `mypkg.3` was installed, the next instance to be added would be named `mypkg.2`.

When asked for *pkgid* in any of the procedures described in this chapter, you must use the package identifier. Remember that when you have only one instance of a package on a machine, which is probably the most common situation, the package identifier is the package abbreviation.

Defining Object Classes

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

The installation parameter `CLASSES`, defined in the `pkginfo` file, is a list of classes to be installed (including the `none` class). Objects defined in the `pkgmap` file that belong to a class not listed in this parameter won't be

installed. If the object is a class action script, then `pkgadd` and `pkgrm` will not copy or run the script, respectively. The actions to be performed on a class are defined in a class action script. These scripts are named after the class itself.

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, `none` is safest. The reason for this is to assure that the directories are created before the objects they will contain and also to assure that no attempt is made to delete a directory before it has been emptied.

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

1. Define the objects belonging to `application` in the `prototype` file entry. For example,

```
f manpage /usr/share/man/man1/myappl.11
f application /usr/bin/myappl
```

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

```
CLASSES=manpage application none
```

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

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

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 installation machine (the default `pkgadd` behavior).

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

Defining Package and Object Locations

Package objects can be delivered either with fixed or relocatable locations.

- Fixed location

Location on the installation machine is defined by the package and cannot be changed. Locations are indicated by specifying absolute pathnames in the `prototype` file.

- Relocatable

There are no absolute path location requirements on the installation machine. The location for relocatable package objects is determined during the installation process.

You can define two types of relocatable objects: *collectively relocatable* and *individually relocatable*. Collectively relocatable objects are located relative to the same directory once the relocatable root directory is established. Individually relocatable objects are not restricted to the same directory location as collectively relocatable objects. Individually relocatable objects should be kept to a minimum since they are more difficult to manage than collectively relocatable objects. If the package contains many different relocations, multiple packages should be considered, each with a different `BASEDIR`.

Defining Collectively Relocatable Objects

Follow these steps to define package objects as collectively relocatable:

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

Put a definition for the `BASEDIR` parameter in your `pkginfo` file. This parameter names a directory where relocatable objects are stored by default. If you supply no value for `BASEDIR`, no package objects are considered collectively relocatable, and all paths in the `pkgmap` must be absolute.

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

You define an object as collectively relocatable by using a relative path name in its entry in the `prototype` file. A relative path name does not begin with a slash. For example, `src/myfile` is a relative path name, while `/src/myfile` is an absolute path name.

Note – A package can deliver some objects with relocatable locations and others with fixed locations.

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

- The value of `BASEDIR` as it is defined in the installer's `admin(4)` file (the `BASEDIR` value assigned in the `admin` file overrides the value in the `pkginfo` file)
- The installer's response to `pkgadd` when asked where relocatable objects should be installed
- The value of `BASEDIR` as it is defined in your `pkginfo` file (this value is used as the default in case the other two possibilities do not supply a value). For interactive installations, this is the most common source of the `BASEDIR`.
- For 2.5 applications and above, the `BASEDIR` can be modified by the `request` script or the `checkinstall` script.

Defining Individually Relocatable Objects

You define a package object as individually relocatable by using a variable in its path name definition in the `prototype` file. Your `request` script must ask the installer where such an object should be stored, then assign the response value to the variable. At the time of installation, `pkgadd` expands the path name based on the output of your `request` script. Case Study 1 in Appendix B, "Packaging Case Studies," shows an example of the use of variable path names and the `request` script needed to solicit a value for the base directory.

This approach may result in widely scattered package components which may be difficult to isolate when installing multiple versions or architectures. Try to use the `BASEDIR` whenever possible.

Writing Your Installation Scripts

Refer to “Installation Scripts” on page 54, for a discussion of scripts that are available to use in installation and how to modify them. You can also look at the case studies in Appendix B, “Packaging Case Studies,” to see examples of ways in which the various scripts can be used.

You are not required to write any installation scripts for a package. The `pkgadd` command performs all the actions necessary to install your package, using the information you supply with the package information files. Any installation script that you provide is used to perform customized actions beyond those executed by `pkgadd`.

Creating the `pkginfo` File

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

```
PARAM=value
```

`PARAM` can be any of the 19 standard parameters described in the `pkginfo(4)` manual page. You can also create your own package parameters simply by assigning a value to them in this file. Your parameter names must begin with a capital letter followed by either upper or lowercase letters.

Parameters beginning with uppercase letters are install time variables. Those beginning with lowercase letters are build time variables. Only install variables become part of the install environment and are visible to all control scripts. Build variables can be used for passing private messages from the `request` script to the `checkinstall` script.

The following five parameters are required:

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

In addition, the `CLASSES` parameter dictates which classes are installed and the order of installation. Although the parameter is not required, no classes will be installed without it. Even if you have no class action scripts, the `none` class must be defined in the `CLASSES` parameter before objects belonging to that class are installed. See also `pkginfo(4)`.

Note – You can choose to define the value of `CLASSES` with a request script instead of defining it in the `pkginfo` file.

You can create the `pkginfo` file with any editor. The following example is for a package that installs a device driver:

```
PKG=bppdev
NAME=bpp device driver
CATEGORY=system
ARCH=sparc
VERSION=bpp release 2.0
CLASSES=none
```

Defining Package Dependencies

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 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 – The reverse dependency type should be used only when a package that cannot deliver a `depend` file relies on the newer package.

- An incompatible package (meaning your package is incompatible with the named package)

Refer to the `depend(4)` and `compver(4)` manual pages for details on the formats of these files.

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

The `depend` file is only a shorthand brute force method for resolving 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 the `request` script or (for 2.5 and above) the `checkinstall` script should be used for this detailed dependency checking. No other control scripts are capable of cleanly halting a `pkgadd`.

Writing a Copyright Message

To deliver a copyright message, you must create a `copyright` file, `copyright`. The message is displayed exactly as it appears in the file (with no formatting) as the package is being installed. Refer to the `copyright(4)` entry in the manual pages for more detail.

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

Reserving Additional Space on the Installation Machine

`pkgadd` ensures that there is enough disk space to install your package, based on the object definitions in the `pkgmap` file. However, 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 during installation. `pkgadd` checks for additional space when you deliver a `space` file with your package. Refer to the `space(4)` manual page for details on the format of this file.

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

Distributing Packages over Multiple Volumes

`pkgmk` performs the calculations and actions necessary to organize a multiple volume package. As `pkgmk` creates your package, it prompts you to insert a new volume as often as necessary to distribute the complete package over multiple volumes. A multiple volume package is called “segmented.”

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 `pkgmk` and forces the placement of the component into the part given in the field. Note that there is a one-to-one correspondence between parts and volumes for removable media formatted as file systems. If the volumes are preassigned by the developer, `pkgmk` will issue an error if there is insufficient space on any volume.

Creating the `prototype` File

Each package must have a `prototype` file that describes the package objects. You can create this file with an editor or with `pkgproto`.

A very common technique is to create the package using `make`. `pkgproto` produces a preliminary `prototype` file which is completed using `awk` or `sed`.

When creating a `prototype` file, you must supply the following information about each object:

- `ftype`
- The object *class*
- The object *pathname*

The path name can define an absolute path name such as `/mypkg/src/filename`, a collectively relocatable path name such as `src/filename`, or an individually relocatable path name such as `$BIN/filename` or `/opt/$PKGINST/filename`.

The format for a `prototype` file entry is as follows:

[part] f_{type} class pathname[major minor][mode owner group]

The path name parameter defines where the component should reside on the installation medium and tells `pkgmk` where to find it on your machine. If these names differ, use the `path1=path2` format for `pathname`, where `path1` is the name it should have on the installation machine and `path2` is the name it has on your machine. Links must use the `path1= path2` format and if the link is relative, it must not begin with a `/`. Refer to the `prototype(4)` manpage for more information on each of the parameters.

Commands may be specified in the `prototype` file to support resolution of file sources and attributes. These commands are:

```
!search
!include
!parameter=value
```

Their scope ranges from the command to the end of the `prototype` file. All `prototype` commands are all local.

All parameters beginning with a lowercase letter are build variables and must be resolved at build time in `pkgmk`. All parameters beginning with an uppercase letter are install variables, and will *not* be resolved at build time. Any global install variable defined at build time will be inserted into the `pkginfo` file for use at install time.

Using an Editor to Create the `prototype` File

You can create a `prototype` file with any editor. There must be one entry for every package component. The following is an example of a `prototype` file:

```
# Package "prototype" file for the bbp device driver.
# Bidirectional Parallel Port Driver for SBus Printer Card.
#
i pkginfo
i request
i copyright
i postinstall
f none bbp.kmod 0444 root sys
f none bbp_make_node 0555 root sys
f none bbp_remove_node 0555 root sys
```

Creating the prototype File with pkgproto

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

`pkgproto` writes its output to the standard output. To create a file, redirect the output to a file. The examples shown in this section do not perform redirection.

Creating a Basic prototype File

The standard format of `pkgproto` is

```
pkgproto [-i] [-c class] [path1[=path2]...]
```

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

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

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

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

Note – If no path names are supplied when executing `pkgproto`, standard input (`stdin`) is assumed to be a list of paths. Refer to the `pkgproto(4)` manual page for details.

Assigning Objects to a Class

You can use the `-c class` option of `pkgproto` to assign objects to a class other than `none`. When using this option, you can only name one class. To define multiple classes in a `prototype` file created by `pkgproto`, you must edit the file after its creation, or call `pkgproto` once per class passing it the package objects via `stdin`.

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

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

***Renaming Pathnames with* `pkgproto`**

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

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

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

`pkgproto` *and Links*

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

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

Creating Links

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

- Its `ftype` 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. Source files can be absolute or relative to the destination. Absolute links must be preceded with a `/`; otherwise, it is considered to be a relative link. For example a `prototype` 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.

Defining Objects for `pkgadd` to Create

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

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

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

The only objects that must be delivered are regular files and edit scripts ($f_{type}=e, v, f$) and the directories required to contain them. `pkgadd` creates the following objects based on the information in the `pkgmap` file without reference to the delivered objects, directories, named pipes, devices, hard links, and symbolic links.

```
d none /directory 644 root other
```

Note that objects cannot be defined with a symbolic link that is a defined object. The reasons can best be illustrated by the following example. Consider the following valid prototype entries:

```
i pkginfo
d none usr 755 root sys
d none usr/bin 755 root sys
s none bin=usr/bin
f none usr/bin/prog=prog 555 root bin
```

After running `pkgmk`, you get a `pkgmap` file that looks something like this:

```
l 65
l s none bin=usr/bin
l i pkginfo 167 13556 651817887
l d none usr 0755 root sys
l d none usr/bin 0755 root sys
l f none usr/bin/prog 0555 root bin 8645 63299 651810096
```

Note how `pkgmk` sorts the `pkgmap` file by pathname, with no regard to the type field in column 1. `pkgmk` also creates the spooled package in this order, `pkgadd` installs it in this order, thus directories are created before any references to files which will reside in those directories.

Now, consider the same `prototype` file, with the exception that the pathname prefix for installed files is specified with respect to a symbolic link, which is also a defined object:

```
i pkginfo
d none usr 755 root sys
d none usr/bin 755 root sys
s none bin=usr/bin
f none bin/prog=prog 555 root bin
```

After running `pkgmk`, you get a `pkgmap` file that looks like this:

```
1 s none bin=usr/bin
1 f none bin/prog 0555 root bin 8645 63299 651810096
1 i pkginfo 167 13556 651815761
1 d none usr 0755 root sys
1 d none usr/bin 0755 root sys
```

Note that the `pkgmap` file is sorted by path name. The attempt to create the file `bin/prog` would fail because the directory `usr/bin`, referenced by the symbolic link `bin`, has not been created yet. This restriction applies whether the package is installed at a fixed location or is relocatable.

Using Commands in a `prototype` file

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

- Nest `prototype` files (the `include` command)
- Define directories for `pkgmk` to look in when attempting to locate objects as it creates the package (the `search` command)
- Set a default value for `mode`, `owner`, and `group` (the `default` command). If all or most of your objects have the same values, using the `default` command keeps you from having to define these values for every entry in the `prototype` file.
- Assign a temporary value for variable pathnames to tell `pkgmk` where to locate these relocatable objects on your machine (with `PARAM=value`).

Creating a Package with `pkgmk`

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

To create a package, execute `pkgmk` as follows:

```
$ pkgmk [-d device] [-r rootpath] [-b basedir] [-f filename]
[PARAM=value]
```

In the `pkgmk` command:

- `-d` specifies that the package should be copied onto *device*, which may be a diskette, CD-ROM image, or online package repository
- `-r` requests that the root directory *rootpath* be used to locate objects on your machine
- `-b` requests that *basedir* be prepended to relocatable paths when `pkgmk` is searching for objects on your machine
- `-f` names a file, *filename*, that is used as your `prototype` file
- `PARAM=value` sets environment variables. Variables beginning with lower case letters are resolved at build time. Those beginning with uppercase letters are placed into the `pkginfo` file for use at install time.

`pkgmk` creates a new instance of a package when you specifically assign a new instance name on the `pkgmk` command line.

Other options are described in the `pkgmk(1M)` manual page. For example, executing `pkgmk -d /vol/dev/rfd0/unlabeled` creates a package based on a file named `prototype` in your current working directory. The package is formatted and copied to the diskette in the device `/vol/dev/rfd0/unlabeled`.

Specifying the Location of Package Contents

This section describes situations that might require supplying `pkgmk` with extra information and provides an explanation of how to do so.

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

Use the *path1=path2* pathname format in your `prototype` file.

- You have relocatable objects in your package.

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

Or you can use the `search` command in your `prototype` file to tell `pkgmk` where to look for objects.

Or you can use the `-b basedir` option of `pkgmk` to define a path name to prepend to relocatable object names while creating the package. For example, the following command looks in the directory `/usr2/myhome/reloc` for any relocatable objects in the package.

```
$ pkgmk -d /vol/dev/rfd0/unlabeled -b /usr2/myhome/reloc
```

- You have variable object names.

You can use the `search` command in your `prototype` file to tell `pkgmk` where to look for objects.

Or you can use the `PARAM=value` command in your `prototype` file to give `pkgmk` a value to use for the object name variables while it creates your package.

Or you can use the `VARIABLE=value` option on the `pkgmk` command line to define a temporary value for variable names.

Note – In determining search paths and include files both build and install variables are resolved. Only build variables will be resolved within the resulting package. For example:

```
!THEdir=/home/buddy/etc
!MYDIR=/home/buddy/src
!thedir=/opt
!search=$MYDIR,$thedir
f none $MYDIR/reader=$THEdir/reader
f none $thedir/staff
f none writer
```

will result in the following `pkgmap` entries:

```
1 f none $ MYDIR/reader
1 f none /opt/stuff
1 none writer
```

-
- The root directory on your machine differs from the root directory described in the `prototype` file.
 - `-r rootpath` ignores destination paths in the prototype files. Instead, it uses the indicated `rootpath` with the source pathname appended to locate objects on the source machine.
 - If you put package information files (such as `pkginfo` and `prototype`) and the package objects in two different directories, indicate this by using the `-b <basedir>` and `-r <rootpath>` options to `pkgmk`. 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
```

Or, you could change directory to the package source directory and use the following command:

```
$ pkgmk -b /product/pkgbin
```

In this case, `pkgmk` uses the current working directory to find the other parts of the package (like the `prototype` and `pkginfo` package information files).

The following example shows the directory structure for a test package and the commands to use for this directory structure:

```
% pwd
/pkgctest
% ls pkgsrc
Makefile
packages/
prototype
pkginfo
SCCS/
copyright
depend
pkgicon
% ls pkgbin
colortool*
cdplayer*
mailcheck*
% more pkgsrc/prototype
#ident "@(#)prototype 1.1      90/09/27 ABC, Inc."
i pkginfo
i pkgicon
i copyright
i depend
f none colortool 0555 user staff
f none cdplayer 0555 user staff
f none mailcheck 0555 user staff
% cd pkgsrc
% pkgmk -o -b /pkgctest/pkgbin -f prototype
--or--
% pkgmk -o -r /pkgctest/pkgsrc -b /pkgctest/pkgbin -f prototype
```

Transferring the Package to the Media with `pkgtrans`

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

- File system format to datastream format
- Data stream format to file system format
- One file system format to another file system format

To perform a basic translation, execute the following command:

```
$ pkgtrans device1 device2 [pkg1[,pkg2[ ...]]]
```

where

- *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[,pkg2...]* is one or more package names

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 *pkg*.

Creating a Datastream Package

Creating a datastream package requires two steps:

1. Create a directory format package using `pkgmk`.

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

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

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

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

```
$ pkgmk -d spool -l 1400
$ pkgtrans spool ctape1 package1
```

The first step formats a package into a fixed directory format under the device alias named `spool`. `spool` is a device alias that exists in `/etc/device.tab`.

The second step translates the fixed directory format of `package1` residing on `spool` into a format supported by the destination device, `ctape1`. If the destination device supports a file system, the translation is in fixed directory format. Otherwise, the translation is in datastream format, which would normally be the case for a device called `ctape1`, a tape device.

The following command is similar to the second step above, except that it sets the datastream package on the medium in a device named `diskette`. `pkgtrans` forces translation into datastream format on the destination device.

```
$ pkgtrans -s spool diskette1 package1
```

The `-s` option indicates that the package will be copied as a datastream. Because a diskette supports a file system format, and `pkgtrans` creates a file system by default (if possible), the `-s` option is necessary to override the default.

Translating a Package Instance

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

Installation Scripts

This section discusses the 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 packaging scripts. However, you may want to create customized installation procedures for your package.

An installation script must be executable by `sh`. It should be made `0644` containing Bourne commands text. There are four types of installation scripts with which you can perform customized actions under Solaris 2.5:

- `request` script

Solicits administrator interaction during package installation for the purpose of assigning or redefining environment parameters.

- Procedure scripts

Specify a procedure to be invoked before or after the installation or removal of a package. The four procedure script types are `preinstall`, `postinstall`, `preremove`, and `postremove`.

- 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 three standard classes (`sed`, `awk`, and `build`). See the section “Creating the prototype File” on page 42 for details on how to define a class.

- `checkinstall` script

Examines the file system for special features and determines whether the install proceeds.. This is only executed on Solaris 2.5 and later.

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, `pkgadd` performs the following steps:

- 1. Executes the `request` script. This is the only point at which your package can solicit input from the installer.**

Note that the `request` script is executed only in interactive mode. In non-interactive mode, the `request` script should have been previously run by `pkgask`. The `request` script is executed in non-privileged mode as user “install.” If no such user is located it is executed as user “nobody.”

- 2. Executes the `checkinstall` script (only under Solaris 2.5 and above.)**
The `checkinstall` script...

3. Executes the `preinstall` script.

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

5. Installs the regular files (`ftype e,v,f`).

Installation occurs class by class. Class action scripts are executed accordingly. The class Action script is only passed regular files to install. All other package objects are created automatically from information in the `pkgmap`. 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 can change the value of `CLASSES`.

6. Creates all hard links.

7. Executes the `postinstall` script.

Script Processing During Package Removal

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

1. Executes the `preremove` script.

2. Removes hard links.

3. Removes regular files.

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 at installation.

4. Removes symbolic links, devices, and named pipes.

5. Executes the `postremove` script.

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

Installation Parameters

The following four groups of parameters are available to all installation scripts. Some of the parameters can be modified by a request script.

- The four system parameters that are part of the installation software (see Table 2-4). None of these parameters can be modified by a package.
- The 19 standard installation parameters defined in the `pkginfo` file. Of these, the `request` or `checkinstall` script can modify only the `CLASSES` parameter. (The standard installation parameters are described in detail in the `pkginfo` entry in the manual pages.) In Solaris 2.5 and above, the `BASEDIR` parameter may be modified.
- You can define your own installation parameters by assigning values to them in the `pkginfo` file. Such parameters must be alphanumeric with initial capital letters. Any of these parameters can be changed by a `request` or `checkinstall` script.
- Your request script can define new parameters by assigning values to them and putting them in the installation environment.

Table 2-4 Installation Parameters

Parameter	Description
PATH	Specifies the search list used by <code>sh</code> to find commands on script invocation, <code>PATH</code> is set to <code>/sbin:/usr/sbin:/usr/bin:/usr/sadm/ install/bin</code>
UPDATE	Indicates that the current installation is intended to update the system. Automatically set to <code>yes</code> if the package being installed is overwriting a version of itself.
PKGINST	Specifies the instance identifier of the package being installed. If another instance of the package is not already installed, the value is the package abbreviation. Otherwise, it is the package abbreviation followed by a suffix, such as <code>pkg.1</code>
PKGSAV	Specifies the directory where files can be saved for use by removal scripts or where previously saved files can be found.

Getting 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 only for the parameters requested.

See the `pkginfo(1)` and `pkgparam(1)` manual pages for details on these tools.

Exit Codes for Scripts

Each script must exit with one of the exit codes shown in Table 2-5.

Table 2-5 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	Script was interrupted and possibly left unfinished. Installation terminates at this point. If <code>checkinstall</code> script returns this, the <code>pkgadd</code> halts cleanly.
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 Appendix B, “Packaging Case Studies,” for examples of exit codes in installation scripts.

Note – All installation scripts delivered with your package should have an entry in the `prototype` file. The file type should be `i`.

The Request Script

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

It is executed as non-privileged user “install” if such a user is identified on the system. Otherwise, it executes as non-privileged user “nobody.”

The output of a request script must be a list of parameters and their values. This list can include any of the parameters you created in the `pkginfo` file and the `CLASSES` parameter. The list can also introduce parameters that have not been defined elsewhere. Remember the binding rules for `install` vs. `build` time parameters.

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

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

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

Request Script Usage Rules

- The request script cannot modify any files. It only interacts with users and creates a list of parameter assignments based upon that interaction. (To enforce this restriction, the request script is executed as the nonprivileged user `install` if that user exists; otherwise it is executed as the nonprivileged user “nobody.”)

- `pkgadd` calls the request script with one argument that names the script's output file.
- The parameter assignments should be added to the installation environment for use by `pkgadd` and other packaging scripts by writing them to `$1`.
- System parameters and standard installation parameters, except for the `CLASSES` parameter, cannot be modified by a request script. Any of the other available parameters can be changed.
- The format of the output list should be `PARAM= value`. For example:

```
CLASSES=none class1
```

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

The Procedure Script

The procedure script gives a set of instructions performed at particular points in installation or removal. Four possible procedure scripts are described below. Appendix B, "Packaging Case Studies," shows examples of procedure scripts.

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

- `preinstall` (executes before class installation begins, no files can be installed by this script)
- `postinstall` (executes after all volumes have been installed)
- `preremove` (executes before class removal begins; no files can be removed by this script)
- `postremove` (executes after all classes have been removed)

Procedure Script Usage Rules

- Procedure scripts are executed as `uid=root` and `gid=other`.
- 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 which installs a package object not in the `pkgmap` 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` may install package objects in this way. (See the entry for the `installf` command in the manual pages and the case studies in Appendix B, “Packaging Case Studies,” for details and examples.)
- User interaction is not permitted during execution of a procedure script. All user interaction is restricted to the request script.
- Each procedure script which removes files not installed from the `pkgmap` 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 the entry for the `removef` command in the manual pages and the case studies in Appendix B, “Packaging Case Studies,” for details and examples.)

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

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 Appendix B, “Packaging Case Studies,” 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
<code>i.class</code>	Operates on pathnames in the indicated class during package installation.
<code>r.class</code>	Operates on pathnames in the indicated class during package removal.

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

Class Action Script Usage Rules

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

Note – The installation service relies upon this condition being met.

- The script is not executed if no files in the given class exist on the current volume.
- `pkgadd` and `pkgrm` 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 pathnames defined in the `pkgmap` that belong to a particular class.
- 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 `pkgadd`.
- When the class action script is executed for the last time (meaning the input path name is the last path on the last volume containing a file of this class), it is executed with the keyword argument `ENDOFCLASS`. This flag enables you to include post-processing actions into your script.

- User interaction is not permitted during execution of a class action script. All user interaction is restricted to the request script.

Installation of Classes

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

1. `pkgadd` creates a pathname list.

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

- The list contains only pathnames belonging to the associated class.
 - Directories, named pipes, character/block devices, and symbolic links are included in the list with the source path name set to `/dev/null`, only under the rare situation that the attempt to create the package object failed. Normally they will be automatically created by `pkgadd` (if not already in existence) and given proper attributes (mode, owner, group) as defined in the `pkgmap` file.
 - Linked files where `ftype` is `l` are not included in the list. `ftype` defines the file type and is defined in the prototype file. Links in the given class are created in Step 4.
 - If a path name already exists on the target machine and it is the same as the path name being installed, it is not included in the list. The path name is judged the same if size and checksum are identical.
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 Step 1. If this is the last volume of the package and there are no more objects in this class, the script is executed with the single argument of `ENDOFCLASS`.

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

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

Removal of Classes

Objects are removed class by class. Classes that exist for a package but 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 steps outline the system actions that occur when a class is removed:

1. `pkgrm` creates a pathname list.

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

Referenced pathnames may be modified to remove information put in it by the package being removed.

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

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

Note – Always assign a class to files with an `ftype` of `e` (editable) and have an associated class action script for that class. Otherwise, the files will be removed at this point, even if the pathname is shared with other packages.

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

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

4. `pkgrm` performs an audit.

After successfully executing the class action script, `pkgrm` removes knowledge of the pathnames from the system unless a pathname is referenced by another package.

The Special System Classes

The system provides three special classes. They are:

- The `sed` class

Provides a method for using `sed` instructions to edit files upon installation and removal.

- The `awk` class

Provides a method for using `awk` instructions to edit files upon installation and removal.

- The `build` class

Provides a method to dynamically construct a file during installation.

The `sed` Class Script

The `sed` installation class provides a method to install and remove objects that modify an existing object on the target machine. A `sed` class action script delivers `sed` instructions in the format shown in the next figure.

Two commands indicate when instructions should be executed. `sed` instructions that follow the `!install` command are executed during package installation and those that follow the `!remove` command are executed during package removal. It does not matter which order the commands are used in the file.

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

```
# 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]
```

address, *function*, and *arguments* are as defined in the `sed(1)` manual page. See Case Studies #4a and #4b in Appendix B, “Packaging Case Studies,” for examples of `sed` class action scripts.

The awk Class Script

The `awk` installation class provides a method to install and remove objects that modify an existing object on the target machine. 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 following figure.

Two commands indicate when instructions should be executed. `awk` instructions that follow the `!install` command are executed during package installation, and those that follow the `!remove` command are executed during package removal. It does not matter in which order the commands are used in the file.

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

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

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

The build Class Script

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


The name of the instruction file should conform to standard system naming conventions.

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

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

See Case Study #4c in Appendix B, "Packaging Case Studies," for an example `build` class action script.

Installing and Checking Packages

3 

This chapter describes how to install and check your software package. You should install from your CD-ROM image and verify that the installation is correct before having CD-ROMs manufactured. The following topics are discussed:

- Installation software database

Describes the database that keeps track of the packages that have been installed.

- Installing software packages

Briefly describes the installation command `pkgadd`. Installing software for clients on a server is also discussed.

- Checking installation accuracy and displaying information about installed packages

Describes how to use the `pkgchk` command to check the integrity of your packages after they have been installed. Also describes the various types of information you can display with the `pkginfo` command.

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 pathname, where the component resides and the component type. Entries are added and removed automatically by `pkgadd` and `pkgrm`. You can view the information in the database by using the `pkgchk` command.

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 `pkgadd` 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.

You can use the `pkginfo` command to survey the contents of the installation software database. The commands `installf` and `removef` can be used to modify its contents.

Installing Software Packages

The default installation mode is interactive. To install a software package named `pkgA` from a disk device named `/dev/dsk/c0t0d0s0`, you would enter the following command:

```
# pkgadd -d /dev/dsk/c0t0d0s0 pkgA
```

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

```
# pkgadd -d /dev/dsk/c0t0d0s0 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, installation fails. The name given after the `-d` option must be a full pathname to a device, directory (as shown in the example), or device alias.

Note – You must use a package identifier if multiple versions reside on the installation medium. In most cases, there is only one instance of a package on a medium and the package identifier is the package abbreviation without a suffix.

Be aware that the suffix of a package identifier defines the package instance on that particular medium. A new package identifier is assigned to this package when it has been installed on your system. (Use `pkginfo -d device` to find out what instances are on a medium.)

Interacting with `pkgadd`

When `pkgadd` encounters a problem, it first checks the `admin` file for instructions. (See the `admin(4)` manual page for details on the format of this file.) If no instructions exist, or if the parameter is defined as `ask`, `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 for Clients on a Server

This section describes how to install packages for a client that place files in the `root` file system. Packages that do not place files in `root` can be made available to clients by installing the package on the server with `pkgadd`. These packages are then made available when the file systems are mounted by the clients.

Unbundled software packages should be installed into `/opt/PKG`. However, some packages, such as a package containing a device driver, must be installed into `/` or `/usr`.

Installing Packages on a Server for Diskless Clients

You use `pkgadd` on a server to install software either for the use of clients. Software installed for the use of clients is installed in the client's `root` file system, not the server's `root` file system.

A diskless client's `root` file system is located on the server, in the directory `/export/root/client`. The client's `/usr` file system is located in `/export/exec/os_identifier/usr`, where `os_identifier` is a string that identifies the operating system, version, and instruction architecture of the client.

Use the `pkgadd` command with the `-R` option to specify the location of the client's root filesystem for the installation. For example:

```
# /usr/sbin/pkgadd -d device -R root_path
```

Files installed in the client's `root` file system appear in the client's software database as `installed`. Files that the client expects to find in its `/usr` file system are shown as `shared` in the client's database. The shared files must be installed on the server with a separate invocation of `pkgadd`.

You can use the `-R` option with other package commands, for example:

```
# /usr/sbin/pkgchk -R root_path
# /usr/sbin/pkgrm -R root_path
```

Installing Packages on a Server for Dataless Clients

Installing a dataless client is similar to the installation procedure covered under “Installing Packages on a Server for Diskless Clients.” The client’s `root` file system must be a remote mount on the server. The client must export its root file system with read and write access to the server’s root.

After mounting the client’s root filesystem on the server, use the `pkgadd` command with the `-R` option to specify the root filesystem of the client for the installation. For example:

```
# /usr/sbin/pkgadd -d device -R root_path
```

Files installed into the client’s `root` file system appear in the client’s software database as `installed`. Files that the client expects to find in the `/usr` file system are shown as `shared` in the client’s database. The shared files must be installed on the server with a separate invocation of `pkgadd`.

Checking Installation Accuracy

`pkgchk (1M)` enables you to check the accuracy of installed files or display information about package files. It checks the integrity of directory structures and the files. `pkgchk` can list or check the following:

- Contents or attributes, or both, of objects currently installed on the system
- Contents of a spooled, uninstalled package
- Contents or attributes, or both, of objects described in the specified `pkgmap` file

For more detailed information about this command, refer to the `pkgchk(1M)` manual page.

`pkgchk` 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.

If you use the `-f` option to `pkgchk`, file attributes are corrected when discrepancies are found.

Displaying Information About Installed Packages

You can use two commands to display information about packages:

- `pkgparam` displays parameter values
- `pkginfo` displays information from the software database

The `pkgparam` Command

`pkgparam` enables you to display the values associated with the parameters you have requested on the command line. The values are retrieved from either the `pkginfo` file for `pkginst` 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.

For detailed information, refer to the `pkgparam` (1) manual page.

The `pkginfo` Command

You can display information about installed packages with the `pkginfo` command. `pkginfo` 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.

Parameter Descriptions for the `pkginfo` Display

Table 3-1 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 3-1 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, <code>system</code> or <code>application</code>).
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 concerning 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, <code>S s 1</code>).
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. This parameter must be set to allow for multiple instances of a package.
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 <code>pkgmk</code> 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, <code>S s 1</code>).

*Table 3-1*Package Parameters

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

You can request that all spooled packages on a particular device, or in a particular directory, be included in the `pkginfo` list by using the `-d` option. For example, the following command shows information in the extracted format for all the packages in the spool directory `/opt/spooldir`:

```
# pkginfo -d /opt/spooldir -x
```

For detailed information about the `pkginfo` command, refer to the `pkginfo(4)` manual page.

The Default `pkginfo` Display

When `pkginfo` 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 int Installation Utilities
system backup Backup/Restore Utilities
application pkgA Package A
application pkgA.2 Package A
application anpkg Another Package
$
```

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. It presents one line of information per package.

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 pkgA anpkg
pkgA Package A
(SunOS) Release 2, Version 3
anpkg Another Package
(SunOS) Release 4
```

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 mypkg
PKGINST: pkgA.3
NAME: Package A
CATEGORY: application
ARCH: SunOS
VERSION: Version 3
INSTDATE: Tue Apr 14 08:41:40 MDT 1988
BASEDIR: /opt/pkgA
VSTOCK: sdr9000
STATUS: completely installed
FILES: 31 installed
           3 linked files
           10 directories
           13 executable
           nnnn blocks used (approx)
SERIALNUM: 201-790b
$
```

Customizing the Contents of the `pkginfo` Display

You can use the `pkginfo` options to specify packages to be included in the display. See the `pkginfo` (1M) manual page for a description of the options.

Creating Icons and Package Clusters

4 

This chapter describes creating icons, package clusters, and how to create them. Clusters can be used with `admintool` to provide a simplified user interface for installing software.

Creating an Icon for Your Application

If you want to provide an icon for your application, you can use `iconedit(1)` to create one. See the `iconedit(1)` man page for more information. Before you save the icon, be sure your `iconedit` settings are as follows:

- Black and White (not color)
- Format is XView Icon
- Size is 64

You set Size and Format using the Properties menu.

You must also specify the location of the icon in the `pkginfo` file. Set the `SUNW_ICON` parameter to the path name of the icon. The path name should be the relative path to the icon.

Using Clusters

A cluster is a group of one or more software objects such that each object is either a software package or another cluster.

Clusters provide a modular view of the contents of the installation medium. By using the `admintool` add software capability, a user can display a list of the clusters contained in a product. The user can then choose which clusters to install.

Clusters can also simplify the removal and upgrade of software packages and the distribution of localization modules and machine-dependent code. You can package each localization module separately, allowing the user to install only the needed modules. You can also put machine-dependent code in individual packages so that only the code for the specific system is installed.

Contents and Order Files

You can use four files to create clusters for your package:

- `.cdtoc` - the CD-ROM table of contents file
- `.clustertoc` - the cluster table of contents file
- `.packagetoc` - the package table of contents file
- `.order` - the package installation order file

These files are optional. You need to use them only if you are using clusters.

Figure 4-1 shows an example of a directory hierarchy for a CD-ROM, including the locations of the contents and order files. A product can consist of multiple clusters, some of which may be optional.

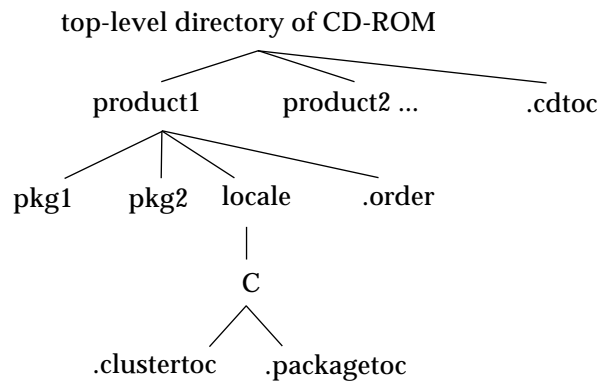


Figure 4-1 CD-ROM Directory Hierarchy Example

If your package has other locales than C (English), there would be a directory under `locale` for each locale. Each locale directory would contain the localized `.clustertoc` and `.packagetoc` files.

`.cdtoc` *File*

The `.cdtoc` resides in the top-level directory on the CD-ROM. It is a text file that describes the location of each product on a CD-ROM. This file is not required to define clusters, but your CD-ROM should contain one if you are putting multiple software products on one CD-ROM.

Each line in the file has the following form:

```
PARAM=value
```

PARAM can be one of:

- `PRODNAME` - full name of the product
- `PRODVERS` - version of the product
- `PRODDIR` - the directory containing the product

The parameters are grouped by product and `PRODNAME` should be the first parameter for each product. See the `cdtoc(4)` man page for detailed information.

`.clustertoc` *File*

The `.clustertoc` file describes all the clusters that make up a product, listing the packages contained in each cluster. This is the only file that is required to define clusters for a product.

The `.clustertoc` file must be in the `locale` directory for the product. Each product can have one `.clustertoc` file describing all the clusters in the product.

Each line in the file has the following format:

```
PARAM=value
```

Parameters are grouped by cluster with the first parameter for a group being `CLUSTER=identifier` and the last parameter for a group being `END`. See the `clustertoc(4)` man page for a complete description of the parameters.

`.packagetoc` *File*

The `.packagetoc` file describes the packages that make up a product. This file must also be in the appropriate subdirectory of the `locale` directory for the product. There can only be one `.packagetoc` per product.

Note – If your product has a `.packagetoc` file, it must also have a `.order` file.

Each line in the file has the following form:

```
PARAM=value
```

Parameters are grouped by package with the first parameter for a package being `PKG=identifier`. See the `packagetoc(4)` man page for a complete description of the parameters.

`.order` *File*

The `.order` file is a text file that specifies the order in which the packages must be installed. The file consists of a list of package identifiers, one per line. The `.order` file resides in the product directory (the directory specified for the product in the `.cdtoc` file). This file is not required to use package clusters. There can be only one `.order` file per product.

Packaging Guidelines



This appendix provides a list of criteria to use when building 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.

Optimize for Client-Server Configurations

You should consider the various types of system software configurations (diskfull, diskless, 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 dataless and 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 to disperse the code into more packages than necessary.

Package by Machine Dependencies

Keep machine 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, SPARC binaries would be in one package and binaries for an Intel machine would be in another.

Overlap in Packages

When constructing the packages, ensure that duplicate files are eliminated when possible. Unnecessary duplication of files results in support and version difficulties. If your product has multiple packages, constantly compare the contents of these packages for redundancies.

Sizing Considerations

Size is package-specific and depends on other criteria. For example, the maximum size of `/opt` should be considered. When possible, a good package should not contain only one or two files or contain extremely large numbers of files. There are cases where a smaller or larger package might be appropriate to satisfy other criteria.

Localization Software Packaging Guidelines

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 or product boundaries criterion.

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.

Introduction

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

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

Case #1: Using a request Script

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

Techniques

This case study shows examples of the following techniques:

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

Approach

To set up selective installation, you must:

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

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

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

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

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

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

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

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

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

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

The request script for this package asks two questions:

- Should this part of the package be installed?

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

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

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

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

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

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

Sample Files

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

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

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

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

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

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

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

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

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

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

Techniques

This case study shows examples of the following techniques:

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

Approach

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

- Create three classes.

This package requires the following three classes to be defined in the `CLASSES` parameter:

- The standard class of `none` (contains a set of processes belonging in the subdirectory `bin`)
 - The `admin` class (contains an executable file `config` and a directory containing data files)
 - The `cfgdata` class (contains a directory)
- Make the package collectively relocatable.

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

- Calculate the amount of space the database file requires and create a `space` file to deliver with the package. This file notifies `pkgadd` that the package requires extra space and specifies how much extra space.
- Create an installation class action script for the `admin` class.

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

- Copies the source data file to its proper destination

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

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

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

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

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

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

Sample Files

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

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

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

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

The following example file shows the space file:

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

The following example file shows the installation script:

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

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

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

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

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

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

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

Techniques

This case study shows examples of the following techniques:

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

Approach

To meet the requirements in the description, you must:

- Create a `copyright` file.

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

- Create a `compver` file.

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

- Create a `depend` file.

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

Sample Files

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

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

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

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

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

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

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

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

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

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

Techniques

This case study shows examples of the following techniques:

- Using the `sed` class

- Using a `postinstall` script

Approach

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

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

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

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

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

- Create a `sed` class action script.

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

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

- Create a `postinstall` script.

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

```
init q
```

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

Sample Files

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

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

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

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

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

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


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

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

Case #4b: Using Classes and Class Action Scripts

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

Techniques

This case study shows examples of the following techniques:

- Creating classes
- Using installation and removal class action scripts

Approach

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

- Create a class.

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

- Create an `inittab` file.

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

- Create an installation class action script.

Since class action scripts must be multiply executable (meaning you get the same results each time they are executed), you cannot just add the sample text to the end of the file. The class action script performs the following procedures:

- Checks to see if this entry has been added before
- If it has, removes any previous versions of the entry
- Edits the `inittab` file and adds the comment lines so you know where the entry is from
- Moves the temporary file back into `/sbin/inittab`
- Executes `init q` when it receives the end-of-class indicator

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

- Create a removal class action script.

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

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

Sample Files

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

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

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

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

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

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

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

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

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

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

Case #4c: Using the build Class

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

Techniques

This case study shows examples of the following technique:

- Using the build class

Approach

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

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

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

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

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

- Create the `build` file.

The `build` file shown next performs the following procedures:

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

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

Sample Files

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

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

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

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

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

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

Case #5: Using crontab in a Class Action Script

This case study modifies crontab files during package installation.

Techniques

This case study shows examples of the following techniques:

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

Approach

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

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

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

- Create the `crontab` files for the package.

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

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

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

- Determines the user ID.

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

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

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

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

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

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

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

Sample Files

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

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

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

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


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

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

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

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

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

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

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

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

Case #6: Installing a Driver

This package installs a driver.

Techniques

This case study shows examples of the following techniques:

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

Approach

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

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

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

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

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

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

- Create a `postinstall` script.

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

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

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

Sample Files

This example shows the prototype file for Case #6:

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

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

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

This example shows the request script for Case #6:

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

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

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

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

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

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

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

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

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

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

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

The pkginfo File

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

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

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

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

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

The sed Class Script

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

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

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

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

The `postinstall` Installation Script

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

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

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

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

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

The preremove Removal Script

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

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

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

exit
```

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

The copyright File

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

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

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

Creating a Package

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

Organize the Package Objects

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

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

Hierarchical Directory Structure

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

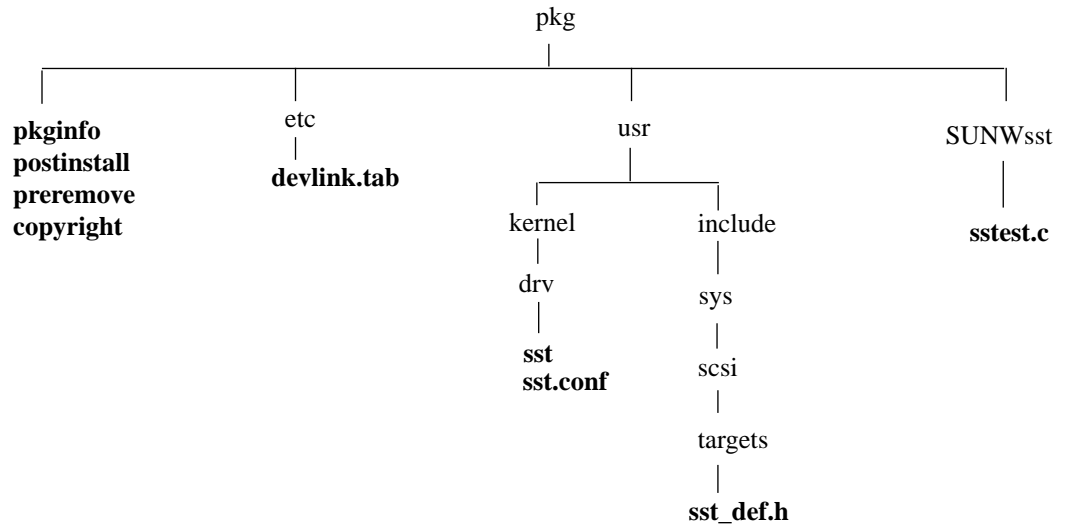


Figure 4-2 Hierarchical Directory Structure

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

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

Flat Directory Structure

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

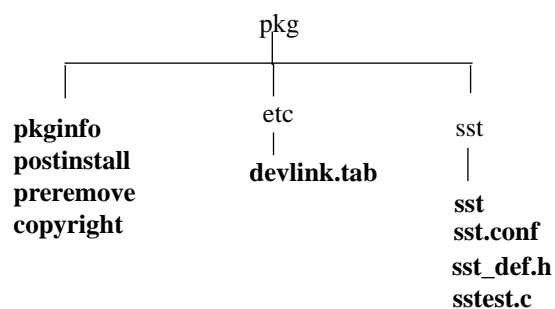


Figure 4-3 Flat Directory Structure

Creating a prototype File for a Hierarchical Directory Structure

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

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

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

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

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

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

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

Creating a prototype File for a Flat Directory Structure

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

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

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

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

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

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

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

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

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

Creating the Package

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

Creating a Package for a Hierarchical Directory Structure

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

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

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

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

Creating a Package for a Flat Directory Structure

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

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

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

The `pkgmap` File

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

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

Transferring the Package to Diskette or Tape

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

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

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

The sed Class Script

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

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

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

`PKG_INSTALL_ROOT`: The root directory of the client.

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

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

Glossary

Artwork

Camera-ready art used to print the CD-ROM disc label, product insert, and J-card. May contain file system type, part numbers, and trademark and copyright information associated with specific product.

ABI

Application Binary Interface.

Caddy

The plastic rectangular container that holds the CD-ROM when it is placed into the CD drive.

Catalyst CDware

The SunSoft Catalyst product that distributes third-party software demos on CD-ROM from SunSoft distribution centers.

CD-ROM

Compact Disc-Read Only Memory.

CD drive

Same as CD player.

CD master (or master)

The glass disc used to create production discs. The master is created by a CD-ROM manufacturer from a magnetic (tape or disc) image sent to them.

CD player

The hardware device used to read CD-ROMs.

Check disc

The name used by some CD-ROM manufacturers to describe a data proof CD-ROM. This may be used to verify that the software product on CD-ROM media is correct.

Coaster

A one-off CD-ROM that has been cut incorrectly.

Copyright

The right to own and sell intellectual property, such as software, source code, or documentation. Ownership must be stated on the CD-ROM and insert text, whether the copyright is owned by SunSoft, or by another party. Copyright ownership is also acknowledged in SunSoft documentation.

DAT

Digital Audio Tape.

dd

The Solaris user command used to copy a file system image to tape.

Disc

The optical compact medium used to hold software or audio data. Discs are read-only, as opposed to read-write.

Disk

The spelling used for hard disk drives, such as SMD, SCSI, or IPI. Does NOT include CD-ROM.

Electronic handoff (EHO)

The term used to describe the method by which an engineering organization hands a CD-ROM product release off to be tested. The product image, as it would appear on CD-ROM, is mounted or copied from one machine to a partition on a test server. Installation is done from this image prior to creating a one-off.

Exabyte

A type of magnetic tape cartridge, 8MM format. It is the 3M vendor-preferred transfer media to receive data for mastering.

First article

The name used to describe a completed CD-ROM. This may be used to verify that the software and disc label on CD-ROM media are correct and can be used to approve a production run of the CD-ROM media. The first article comes

with complete artwork printed on the face of the CD-ROM, including file system format type, part numbers, and trademark and copyright information associated with a specific product release.

High Sierra

Early CD-ROM file system standard. See ISO 9660.

Image tape

The software in its file system format ready to be transferred to a CD-ROM. This image of the software may be transported on a variety of media, 1/2 inch reel magnetic tape, 1/4 inch cassette magnetic tape, Exabyte magnetic tape, or compact disc before being mastered into a CD-ROM.

Insert

The name used to describe the document inside the front of the CD-ROM jewel box. Among other things, the insert may carry product specific information and installation instructions.

ISO 9660

An industry standard file system used for distribution of software on CD-ROM. This file system is tailored to the read-only environment, but does not currently contain many essential POSIX features. This format is often erroneously referred to as High Sierra, which is a subset.

J-card

Printed card with a small, folded edge that fits into the back of the CD-ROM jewel box. It carries the product name and part number so that this information can be read without opening the jewel box. The edge corresponds to the spine of a book.

Jewel box

The plastic case that contains the CD-ROM, the insert, and the J-card.

Mastering fee

The CD-ROM manufacturer charges a mastering fee to cover the expenses of preparing to make a CD-ROM for production. This fee includes downloading your data to a mastering machine, creating a glass master, creating a metal stamper used in the production run, and testing for quality control of these processes.

One-off

A special CD-ROM used as a *check disc*. This CD-ROM is relatively costly and cannot be duplicated. It should be used only for internal testing purposes before mass production.

Packages

A collection of file and directories required for a software application. Packages are created with `pkgmk` or `swmtool`. The software resides in a directory hierarchy which also contains various information files describing the package contents.

Premastering machine

The equipment used for cutting pre-master CD-ROMs. Premastering machine is synonymous with one-off machine. It may be more accurate to call it a CD-Write Once machine, writable CD-ROM publisher, or check disc recorder. Premastering usually describes the whole process previous to mastering.

Production disc

The disc distributed to customers.

Replication

The production run of CD-ROM discs at the manufacturer's facilities.

Rock Ridge

Extensions to the ISO 9660 standard for CD-ROMs. These extensions provide functionality needed by UNIX file systems, such as symbolic links. SunSoft currently uses this file system for distribution of its Solaris software.

tar

Tape Archive Retrieval. Solaris command for adding or extracting files from a media.

UFS

The file system format currently used for Sun's unbundled products on CD-ROM. The same format is used for file systems on disk partitions.

Index

Symbols

.cdtoc file 81
.clustertoc file 81
.order file 82
.packagetoc file 82

A

abbreviation, package 33
artwork, CD-ROM 5
audio data, distributing 9
awk class 65
awk class script 66

B

build class 65
 example 102
build class script 67

C

CD-ROM
 artwork 5

 creating using Rock Ridge 17
 creation task 8
 development cycle 3
 documentation 4
 documentation process 7
 features 1
 insert 5
 installing software from 15
 manufacturing process 11
 packaging 5
 testing 8
check disk 10
checking package installation 73
class action scripts 55, 61
 example 90, 104
 removal example 99
 usage rules 62
classes
 build 67
 build example 102
 installing 63
 removing 64
 sed 65

sed example 96
system 65

classes awk 66

clients
installing packages for 72

cluster
definition 79
uses 80

compact disc *See* CD-ROM

compver file 29, 41
example 94

copyright file 29, 41
example 94, 115

creating a CD-ROM 8

creating a CD-ROM using Rock Ridge 17

creating a package 31

creating a pkginfo file 39

creating a prototype file 42

creating an icon 79

D

data proof 10

depend file 29, 40
example 94

displaying information about installed
packages 74

distributing audio data 9

driver, package example 108

F

file system
High Sierra 12
ISO 9660 12
RockRidge 13
UFS 13

first article 10

G

guidelines, packaging 83

H

High Sierra File System (HSFS) 12

I

icon, creating 79

insert 5

installation
parameters 58
software database 70

installation parameters 57
example 85

installation scripts 29, 39, 54
example 99
exit codes 58
parameters 57
processing of 56

installation software database 70

installation, full 70

installf command example 90

installing a driver, package example 108

installing classes 63

installing packages 71

installing packages on diskless clients 72

installing software from CD-ROM 15

ISO 9660 12
Rock Ridge extensions to 13

J

J-card 5

jewel box 5

L

links, creating with prototype file 46

M

manufacturing process for a CD-ROM 11

O

object classes 35, 45

awk 65

build 65

example 99

installing 63

removing 64

sed 65

system 65

object locations 30

objects 30

relocatable 37

one-off machine 9

P

package abbreviation 33

package components 21, 26

optional 26, 29

required 26, 28

package dependencies 40

package instance 33

package objects 30

classes 35

locations 30

relocatable 37

packages

checking installation 73

creation 31, 49

datastream 53

displaying information about 74

identifier 33, 35

installing 71

installing for clients 72

installing on diskless clients 72

licensing considerations 23

objects 28

relocatable objects 38

software 21

transferring to the media 52

translation 54

packaging guidelines 83

packaging overview 27

pkgadd command 63, 71

pkgchk command 73

pkginfo command 74

pkginfo file 28

creating 39

example 111

pkgmap file 28

pkgmk command 49

pkgparam command 58, 74

pkgproto command 44

and links 46

pkgrm command 64

pkgtrans command 52

postinstall script 60

example 97, 108, 113

postremove script 60

preinstall script 60

pre-mastering machine 9

preremove script 60
 example 108, 114

procedure scripts 55, 60
 usage rules 61

prototype file 28
 creating with an editor 42
 creating with pkgproto command 44
 example 111
 format 42
 using commands in 48

R

relocatable objects 37

removing classes 64

request script 55, 59
 example 85, 109
 usage rules 59

Rock Ridge extensions 13

Rock Ridge, creating a CD-ROM using 17

S

scripts
 awk class 66
 build class 67
 class action 55, 61
 example 90, 104
 removal 99
 usage rules 62
 installation example 99
 postinstall 60
 example 97, 113
 postremove 60
 preinstall 60
 preremove 60
 example 114

procedure 55, 60
 usage rules 61

processing of 56

request 55, 59
 usage rules 59

request example 85

sed class 65
 example 112

sed class 65
 example 96

sed class script 65
 example 112

software licensing 23

software packages 15, 16, 21
 components 21, 26
 creating 31, 49
 instance 33
 object locations 30
 objects 30
 optional components 26, 29
 overview 27
 required components 26, 28

space file 29, 41
 example 90

system object classes 65

T

testing a CD-ROM 8

translating a package 54

U

UFS 13

Copyright 1995 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100 U.S.A.

Tous droits réservés. Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peuvent Être reproduits sous aucune forme, par quelque moyen que ce soit sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il en a.

Des parties de ce produit pourront être dérivées du système UNIX[®], licencié par UNIX System Laboratories Inc., filiale entièrement détenue par Novell, Inc. ainsi que par le système 4.3. de Berkeley, licencié par l'Université de Californie. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

LEGENDE RELATIVE AUX DROITS RESTREINTS: l'utilisation, la duplication ou la divulgation par l'administration américaine sont soumises aux restrictions visées à l'alinéa (c)(1)(ii) de la clause relative aux droits des données techniques et aux logiciels informatiques du DFARS 252.227-7013 et FAR 52.227-19. Le produit décrit dans ce manuel peut être protégé par un ou plusieurs brevet(s) américain(s), étranger(s) ou par des demandes en cours d'enregistrement.

MARQUES

Sun, Sun Microsystems, le logo Sun, Solaris sont des marques déposées ou enregistrées par Sun Microsystems, Inc. aux États-Unis et dans certains autres pays. UNIX est une marque enregistrée aux États-Unis et dans d'autres pays, et exclusivement licenciée par X/Open Company Ltd. OPEN LOOK est une marque enregistrée de Novell, Inc. PostScript et Display PostScript sont des marques d'Adobe Systems, Inc.

Toutes les marques SPARC sont des marques déposées ou enregistrées de SPARC International, Inc. aux États-Unis et dans d'autres pays. SPARCcenter, SPARCcluster, SPARCcompiler, SPARCdesign, SPARC811, SPARCengine, SPARCprinter, SPARCserver, SPARCstation, SPARCstorage, SPARCworks, microSPARC, microSPARC-II, et UltraSPARC sont exclusivement licenciées à Sun Microsystems, Inc. Les produits portant les marques sont basés sur une architecture développée par Sun Microsystems, Inc.

Les utilisateurs d'interfaces graphiques OPEN LOOK[®] et Sun[™] ont été développés par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique, cette licence couvrant aussi les licenciés de Sun qui mettent en place OPEN LOOK GUIs et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit du X Consortium, Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ÉTAT" SANS GARANTIE D'AUCUNE SORTIE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS À RÉPONDRE À UNE UTILISATION PARTICULIÈRE OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

CETTE PUBLICATION PEUT CONTENIR DES MENTIONS TECHNIQUES ERRONÉES OU DES ERREURS TYPOGRAPHIQUES. DES CHANGEMENTS SONT PÉRIODIQUEMENT APPORTÉS AUX INFORMATIONS CONTENUES AUX PRÉSENTES. CES CHANGEMENTS SERONT INCORPORÉS AUX NOUVELLES ÉDITIONS DE LA PUBLICATION. SUN MICROSYSTEMS INC. PEUT RÉALISER DES AMÉLIORATIONS ET/OU DES CHANGEMENTS DANS LE(S) PRODUIT(S) ET/OU LE(S) PROGRAMME(S) DÉCRITS DANS CETTE PUBLICATION À TOUTS MOMENTS.



Adobe PostScript

