Application Packaging 2

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.

2

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 `prototype` file for input to `pkgmk`
`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
`classaction`	
`preremove`	
`postremove`	
`request`	
`checkinstall`	

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.

2

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

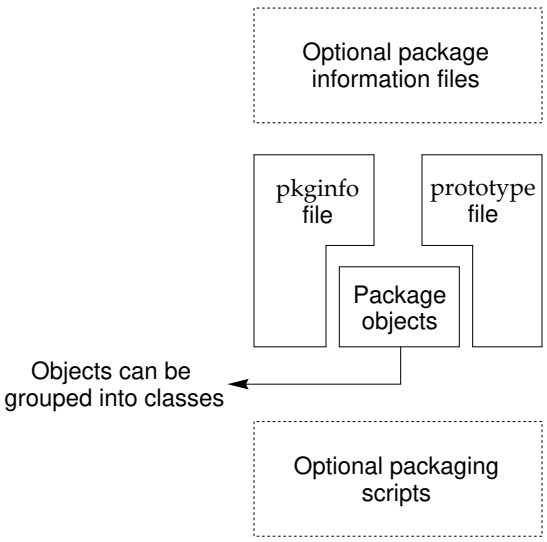

Figure 2-1 The Contents of a Package

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

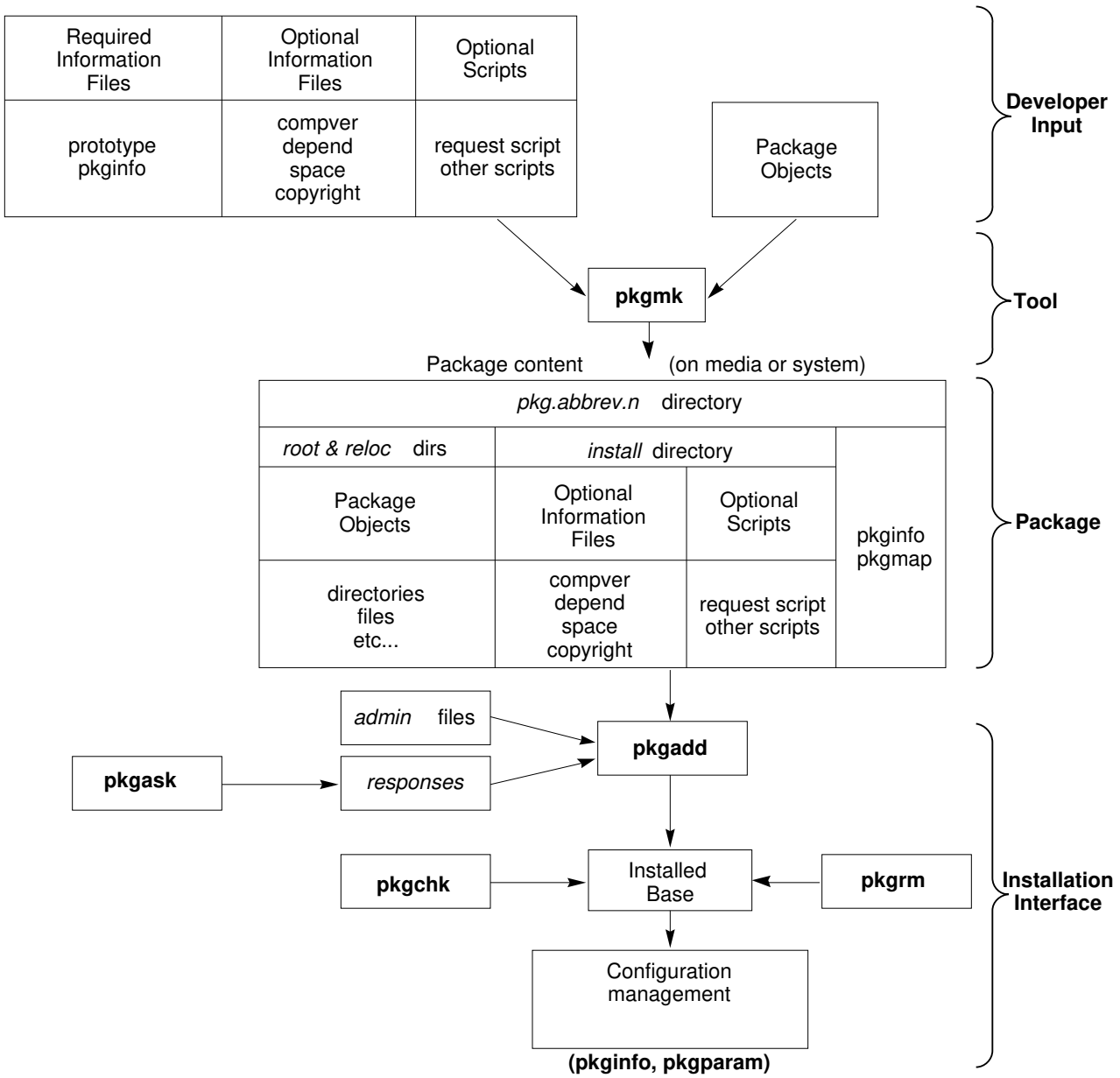

Figure 2-2 Packaging Overview

2

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 to 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**
`PKG=myapp1`	`PKG=myapp1`
`VERSION=1.0`	`VERSION=1.0`
`ARCH=sparc`	`ARCH=intel`

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=myapp1	PKG=myapp1
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/manl/myappl.1l
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] ftype 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
```

2

Creating the `prototype` *File with* `pkgproto`

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

`pkgproto` writes its output to the standard output. To create a file, 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 (`ftype=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
/pkgtest
% 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 /pkgtest/pkgbin -f prototype
   --or--
% pkgmk -o -r /pkgtest/pkgsrc -b /pkgtest/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 `sh` to find commands on script invocation, `PATH` is set to `/sbin:/usr/sbin:/usr/bin:/usr/sadm/install/bin`
`UPDATE`	Indicates that the current installation is intended to update the system. Automatically set to `yes` 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 `pkg.1`
`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 `checkinstall` script returns this, the `pkgadd` 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
`i.`*class*	Operates on pathnames in the indicated class during package installation.
`r.`*class*	Operates on pathnames in the indicated class during package removal.

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

Class Action Script Usage Rules

- 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
# awk1(1) program to remove changes
 . . . (awk program)
```

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

2