# Packaging and Delivering Software with the Image Packaging System

## A developer's guide


2019-05-22:14:55:52-UTC 17bc8e5a

Copyright © 2011, Oracle and/or its affiliates. All rights reserved.
This document is provided for information purposes only and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices.  UNIX is a registered trademark licensed through X/Open Company, Ltd. 0611

# Preface

In OpenIndiana system software is packaged with the Image Packaging System, or IPS. IPS takes care of installing new software and upgrading that software.

This manual is for developers and advanced administrators who want to better understand IPS, how to use it to package their own software, and want to understand how OpenIndiana is packaged with IPS.

Special attention is given to the underlying design concepts and design patterns so that readers can more readily understand and utilize the more advanced features of IPS.

# How this book is organized

Chapter 1 - *Design Goals and Concepts*, outlines the basic design philosophy of IPS and its expression as software patterns.

Chapter 2 - *Package Lifecycle*, provides an overview of the software package lifecycle with IPS.

Chapter 3 - *Basic Terminology*, lays out the basic terminology and describes the various components that form IPS.

Chapter 4 - *Packaging Software with IPS*, gets the new user started constructing their own packages.

Chapter 5 - *Installing, Removing, and Updating Software Packages*, shows basic operation of pkg(1).

Chapter 6 - *Specifying Dependencies*, explains the different types of IPS dependencies and how they can be used to construct working software systems.

Chapter 7 - *Allowing Variations*, explains how variants, facets and mediated links are used to allow software publishers to define multiple installation forms for their packages.

Chapter 8 - *Modifying Package Manifests Programmatically*, explains how package manifests can be machine edited to permit the automated annotation and checking of package manifests.

Chapter 9 - *Causing System Change With SMF*, explains how to use the Service Management Facility to automatically handle any necessary system changes that should occur as a result of package installation.

Chapter 10, *Advanced Update*, deals with more complex package upgrade issues, and describes several features in IPS designed to simplify these problems.

Chapter 11, *Signing Packages*, explains how package signing works and how developers and QA organizations can sign either new or existing, already signed packages.

Chapter 12, *Handling Non-Global Zones*, describes how IPS handles zones and discusses those cases where package developers should be aware of zones.

Chapter 13, *How IPS Features Are Used When Packaging the OpenIndiana OS*, describes how the packages for the operating system are constructed and how the various dependency types in IPS are used to define working package sets.

Chapter 14, *Republishing Packages*, describes how administrators can modify existing packages if needed for local conditions.

Appendix A: *Classifying Packages*, contains info.classification scheme definitions.

Appendix B: *Converting SVR4 packages to IPS*, gives an example of converting an SVR4 package to IPS, and highlights some areas that might need special attention.

# Chapter 1

## Design Goals and Concepts

This chapter discusses IPS design goals and concepts, and discusses some of the implications of those choices.

IPS is designed to eliminate some long-standing issues with previous software distribution, installation and maintenance mechanisms that have caused significant problems for users, developers/maintainers and ISVs.

Principle IPS design concepts and goals include:

- Minimize planned downtime by making software update possible while machines are in production.
- Minimize unplanned downtime by supporting quick reboot to known working software configurations.
- Automate, as much as possible, the installation of new software or updates to existing software.
- Resolve the difficulties with ever-increasing software size and limited distribution media space.
- Ensure that it is possible to determine whether or not a package is correctly installed as defined by the author (publisher) of the package; such a check should not be spoofable.
- Incorporate mechanisms to allow for the easy virtualization of OpenIndiana at a variety of levels - and zones in particular.
- Reduce the effort required to generate patches/upgrades for existing systems.
- Allow other software publishers (ISVs and end-users themselves) to create and publish packages using IPS.

These goals led fairly directly to the following ideas:

- Leverage ZFS snapshot and clone facilities to dynamically create boot environments on an as-needed basis.

  This means that:

  - OpenIndiana requires ZFS as the root file system; zone file systems need to be on ZFS as well.
  - Users can create as many boot environments as desired.
  - The packaging system can automatically create boot environments on an as-needed basis, either for backup purposes prior to modifying the running system, or for installation of a new version of the OS.
- Eliminate duplicated mechanisms and code used to install, patch and update the operating system.

  This results in several significant changes to the way the operating system is maintained. In particular:

  - All OS software updates and patching are done directly with IPS.
  - Any time a new package is installed, it is already exactly at the correct version.
- The requirement for unspoofable verification of package installation has interesting consequences:
  - If a package needs to support installation in multiple ways, those ways must be specified by the developer, so the verification process could take this into account.
  - Scripting is inherently unverifiable since we cannot determine the intent of the script writer. This, along with other issues mentioned later, led to the elimination of scripting during packaging operations.
  - There can be no mechanism for the package to edit its own manifest, since verification is then impossible.
  - If the administrator wants to install a package in a manner incompatible with the original publisher's definition, we should enable the administrator to easily republish the package he wants to alter so that

the scope of his changes are clear, not lost across upgrades, and can be verified in the same manner as the original package.

- The need to avoid size restrictions led to a software repository model, accessed using several different methods. Different repository sources can be composited to provide a complete set of packages, and repositories can be distributed as a single file. In this manner, no single media is ever required to contain all the available software. In order to support disconnected/firewalled operations, tools are provided to copy and merge repositories.
- The desire to enable multiple (possibly competing) software publishers led us to driving all the packaging metadata into the packages themselves, so no master database of all packages, dependencies, etc. exists. A catalog of available packages from a software publisher is part of the repository for performance reasons, but it can be regenerated from the data contained in the packages at will.

# Software Self-Assembly

Given the goals and ideas above, IPS introduces the general concept of *software self-assembly*: Any collection of installed software on a system should be able to build itself into a working configuration when that system is booted, by the time the packaging operation completes, or at software runtime.

Software self-assembly eliminates the need for install-time scripting in IPS. The software is responsible for its own configuration rather than relying on the packaging system to perform that configuration on behalf of the software. Software self-assembly also enables the packaging system to safely operate on alternate images, such as boot environments that are not currently booted, or offline zone roots. In addition, since the self-assembly is performed only on the running image, the package developer does not need to cope with cross-version or cross-architecture run-time contexts.

There are obviously some aspects of preparing an operating system image that must be done before boot, and IPS manages this transparently. These items include updating boot blocks, preparing a boot archive (ramdisk), and on some architectures, managing the menu of boot choices.

Several idioms are employed to facilitate software self-assembly:

- **Actions**

  *Actions* are the atomic units of software delivery in IPS. Each action delivers a single software object - either a file system object, such as a *file*, *directory* or *link*, or a more complex software construct, such as a *user*, *group* or *driver*. These more complex action types, previously handled by SVR4 class action scripts no longer require scripting.

  Actions, grouped together into *packages*, can be installed, updated and removed from both live images as well as offline images.

  While IPS allows for the set of known action types to be extended in the packaging system, during development we have found that the action types delivered at present are sufficient for all packaged software in OpenIndiana. It is not expected that package developers will need to create new action types.

  Actions are discussed in more detail in Chapter 3.

- **Composition**

  Rather than maintaining complex configuration files, that require extensive scripting in order to update each configuration file during packaging operations, IPS encourages package authors to deliver fragments of the complete configuration file.

  The packaged application either accesses those fragments directly when reading its configuration, or the fragments can be assembled into the complete configuration file before reading it.

  A good example of this is the `/etc/user_attr` configuration file, used by OpenIndiana to configure extended attributes for roles and users on the system.

Special service `svc:/system/rbac:default` now can be used to regenerate `/etc/user_attr` from the separate files delivered into the directory `/etc/user_attr.d`. Multiple packages deliver fragments of the complete configuration. IPS action delivering these files are marked with `restart_fmri` attribute wich causes service restart when these fragments are installed, removed or updated. Now no additional scripting is required to update this file.

Obviously this requires that the software is written with composition in mind, which isn't always possible.

- **Actuators & SMF services**

  An *actuator* is a tag applied to any *action* delivered by the packaging system that causes a system change when that action is installed, removed, or updated.

  These changes are typically implemented as SMF services.

  We can create SMF services that are responsible for configuring software directly, or constructing configuration files using data delivered in the SMF manifest or sourced from files installed on the system.

  Since SMF services have a rich syntax to express dependencies, we can ensure that each service only runs when all of its dependencies have been met.

# Designing Your Package

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.

## Naming Your Package

OpenIndiana uses a hierarchical naming strategy for IPS packages. Wherever possible, design your package names to fit into the same scheme. Try to keep the last part of your package name reasonably unique such that `pkg install <name>` doesn't report conflicts.

## Optimize for Client-Server Configurations

You should consider the various patterns of software use (client and server) when laying out packages. Good packaging design divides the affected files to optimize installation of each configuration type. For example, for a network protocol implementation, it should be possible to install the client without necessarily installing the server. Note that if client and server share implementation components, a base package containing the shared bits is necessary.

## Package by Functional Boundaries

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

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

## Package Along License or Royalty Boundaries

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

## Overlap in Packages

Packages that overlap (deliver differing content to the same file system locations, for example) cannot be installed at the same time. Since this error might not be caught until final planning for installation, it can provide a poor user experience, though `pkglint(1)` can help to detect this during the package authoring process.

If the package content must differ, declare an exclude dependency so that IPS will understand that these packages are not to be installed together.

## Sizing Considerations

A package represents (modulo *facets*, discussed later) a single unit of software, and is either installed or not installed. Packages that are always installed together should be combined. Since IPS downloads only changed files on update, even large packages update quickly if change is limited.

# Chapter 2

## Package Lifecycle

This chapter provides an overview of the software package lifecycle with IPS.

Software packages go through a detailed lifecycle with IPS. Understanding the various phases of the package lifecycle will help the developer and administrator optimize their results. The following sections provide a high-level description of each state in the package lifecycle:

### Creation

Packages can be created by anybody. IPS does not impose any particular software build system or directory hierarchy on the part of package authors. More detail about package creation is available in Chapter 4. Aspects of package creation are discussed throughout the remaining chapters of this guide.

### Publication

Packages are published to an IPS repository, either via HTTP or to the file system. If desired, once packages are published they can converted to a `.p5p` package archive file. To access software from an IPS repository, the repository can be added to the system, using the `pkg set-publisher` command, or accessed as a temporary source, using the `-g` flag to `pkg(1)`. Examples of package publication are shown in Chapter 4.

### Installation

Packages can be installed on a system, either from an IPS repository, accessed over http://, https:// or file:// URLs, or installed directly from a `.p5p` package archive. Package installation is described in more detail in Chapter 5.

### Updates

Updated versions of packages might become available, either published to an IPS repository, or delivered as a new `.p5p` package archive.

Installed packages can then be brought up to date, either individually, or as part of an entire system update.

It is important to note that IPS does not use the same concept of "patching" as the SVR4 packaging system did: all changes to packaged software are delivered by updated packages.

The packaging system is optimized to install only the changed portions delivered by an updated package, but essentially, package updates are performed in much the same way as package installs. Package updating is described in more detail in Chapter 5.

### Renaming

During a package's lifecycle, it might be desirable to rename a package. Often this is done for organizational reasons or to refactor packages.

Examples of package refactoring would be where there is an interest in combining several packages into a single package, breaking a single package into multiple smaller packages, or a combination of the two.

IPS gracefully handles actions that move between packages, and has capabilities to allow old package names to persist on the system, automatically installing the new packages when a user asks to install a renamed package. Package renaming is described in more detail in Chapter 10.

## Obsoletion

Eventually a package might reach the end of its life. A package publisher might decide that a package will no longer be supported, and that it will not have any more updates made available. IPS allows publishers to mark such packages as obsolete.

Obsolete packages can no longer be used as a target for most dependencies from other packages, and any packages upgraded to an obsolete version are automatically removed from the system. Package obsoletion is described in more detail in Chapter 10.

## Removal

Finally, a package can be removed from the system assuming that no other packages have dependencies on it. Package removal is described in more detail in Chapter 5.

# Chapter 3

## Basic Terminology

This chapter defines IPS terms and describes the IPS components.

### Image

IPS is designed to install packages in an image. An image is a directory tree, and can be mounted in a variety of locations as needed. Images are of three types:

**Full**

in a full image, all dependencies are resolved within the image itself and IPS maintains the dependencies in a consistent manner;

**Zone**

in a zone image, IPS maintains the zone consistent with its global zone as defined by dependencies in the packages;

**User**

not yet fully functional for OpenIndiana.

In general, images are created or cloned by other software (installers, `beadm(1M)`, `zonecfg(1M)`, etc) rather than directly by the user.

### Package

IPS deals with all software installed on a system in the granularity of packages. Every package is represented by a *fault management resource identifier* (FMRI), consisting of a publisher, a name, and a version, with the scheme '`pkg`'. For example:

```
pkg://openindiana.org/system/library@0.5.11,5.11-2018.0.0.18233:20190417T022656Z
```

Here, '`openindiana.org`' is the publisher, '`system/library`' is the package name, and '`0.5.11,5.11-2018.0.0.18233:20190417T022656Z`' is the version.

Package names are hierarchical with an arbitrary number of components separated by forward slash ('`/`') characters. Package names form a single namespace across publishers; packages with the same name and version but different publishers are assumed to be interchangeable in terms of external dependencies and interfaces. Package name components are case sensitive and must start with a letter or number, but can include underscores ('_'), dashes ('`-`'), periods ('`.`'), and plus signs ('`+`') in later positions.

FMRIs can appear and can be referred to in abbreviated form. The scheme is typically unnecessary, leaving the FMRI to start with either a double slash ('`//`') or a single slash ('`/`'). When the first slash is doubled, the first word following the slash is the publisher name. When there is only a single leading slash, no publisher name is present, and the package name is considered complete, or 'rooted'.

Further abbreviation is possible by eliding leading components of package names. For instance, `/driver/network/e1000g` can be reduced to `network/e1000g`, or even simply `e1000g`. Note that such abbreviation mighy cause the packaging client to complain about ambiguous package names, in which case disambiguation can always be achieved by specifying the full, rooted name. Typically package names are chosen to reduce possible ambiguities, even when referred to solely by their last component. Some trailing components are common, however; in such cases, the last two components should be unambiguous. Scripts should generally refer to packages by their full, rooted names.

It is not possible to construct an abbreviated FMRI that contains a publisher name and only trailing package name components.

The version is also often unnecessary; packages referred to without version will generally resolve to the latest version of the package that can be installed. As explained below, versions themselves need not be complete.

FMRIs can also be referred to with patterns, where an asterisk (‘`*`’) can match any portion of a package name. Thus `/driver/*/e1000g` will expand to `/driver/network/e1000g`, as will `/dri*00g`.

### Version

A package version consists of four sequences of integer numbers, separated by punctuation. The elements in the first three sequences are separated by dots, and the sequences are arbitrarily long. Leading zeros in version components (e.g. ‘`01.1`’ or ‘`1.01`’) are forbidden, to allow for unambiguous sorting by package version.

An example version is:

```
0.5.11,5.11-2018.0.0.18233:20190417T022656Z
```

The first part is the component version. For components that are are provided by illumos-gate, this is the OS major.minor version. For a component with its own development life cycle, this sequence is the dotted release number, such as ‘`2.4.10`’.

The second part, which if present must follow a comma, is the build version. OpenIndiana uses this to denote the release of the OS for which the package was compiled.

The third part, which if present must follow a dash, is the branch version, providing vendor-specific information. This can be incremented when the packaging metadata is changed, independently of the component; can contain a build number; or provide some other information.

The fourth part, which if present must follow a colon, is a timestamp. It represents when the package was published in the GMT timezone, and is automatically updated when the package is published.

The package versions are ordered using left-to-right precedence; thus the timestamp is the least significant part of the version space; the number immediately after the ‘`@`’ is the most significant.

If required, `pkg.human-version` can be used to hold a human-readable version string, however the versioning scheme described above must also be present. The human-readable version string is only used for display purposes, and is documented later in this chapter.

By allowing arbitrary version lengths, IPS can accommodate a variety of different models for supporting software. Within the confines of a given component version, a package author can use the build or branch versions and assign one portion of the versioning scheme to security updates, another for paid vs. unpaid support updates, another for minor bug fixes, etc.

A version can also be the token ‘`latest`’, which is substituted for the latest version known.

We discuss how OpenIndiana implements versioning in Chapter 13.

## Publisher

A publisher is an entity that develops and constructs packages. A publisher name, or prefix, is used to identify this source in a unique manner. The use of Internet domains or registered trademarks is encouraged, since it provides a natural namespace partitioning.

Package clients combine all specified sources of packages for a given publisher when computing packaging solutions. Publisher names can include upper and lower case letters, numbers, dashes and periods; the same characters as a valid hostname.

## Action

Actions are used to define the software that comprises a package; they define the data needed to create this software component. When creating packages, the developer expresses the package contents as a set of actions then saves those to a *package manifest* file.

Actions look like this:

```
action_name attribute1=value1 attribute2=value2 ...
```

As a concrete example:

```
dir path=a/b/c group=sys mode=0755 owner=root
```

The first field identifies this as a `dir` (or directory) action; the `name=value` attributes describe the familiar properties of that directory. In the cases where the action has data associated with it, such as a file, the action looks like this:

```
file 11dfc625cf4b266aaa9a77a73c23f5525220a0ef path=etc/release owner=root \
    group=sys mode=0444 chash=099953b6a315dc44f33bca742619c636cdac3ed6 \
    pkg.csize=139 pkg.size=189 variant.arch=i386
```

Here the second attribute (without a `name=` prefix), called the payload, is the SHA-1 hash of the file. This attribute can alternatively appear as a regular attribute with the name `hash`; if both forms are present they must have the same value.

Action metadata is freely extensible; additional attributes can be added to actions as desired. Attribute names cannot include spaces, quotes, or equals signs (‘=’). Attribute values can have all of those, although values with spaces must be enclosed in single or double quotes. Single quotes need not be escaped inside of a double-quoted string, and vice versa, though a quote can be prefixed with a backslash (‘\’) so as not to terminate the quoted string. Backslashes can be escaped with backslashes. It is recommended that custom attributes use a reverse domain name or similar unique prefix to prevent accidental namespace overlap.

Multiple attributes with the same name can be present and are treated as unordered lists.

Note that manifests are largely created using programs; it is not expected that that developers produce complete manifests by hand, but rather create skeletons with the minimal non-redundant information, and have the rest filled in with tools such as `pkgmogrify(1)` and `pkgdepend(1)`.

Most actions have key attributes; this attribute is what makes this action unique from all others in the image. For file system objects, this is the path for that object.

## Types of Actions

There are currently twelve action types in IPS. The following sections describe each action type, and the attributes that define these actions. The action types are detailed in the `pkg(5)` man page, and are repeated here for reference.

Each section contains an example action, as it would appear in a manifest during package creation. Other attributes might be automatically added to the action during publication.

### File Actions

The `file` action is by far the most common action, and represents an ‘ordinary file’. The file action references a payload, and has four standard attributes:

**path**

The file system path where the file is installed. This is a file action's key attribute. These are relative to the root of the image.

**mode**

The access permissions (in numeric form) of the file. These are simple permissions only, not ACLs.

**owner**

The name of the user that owns the file.

**group**

The name of the group that owns the file.

The payload is a positional attribute in that it is not named. It is the first word after the action name. In a published manifest, it is the SHA-1 hash of the file contents. If present in a manifest that has yet to be published, it represents the path where the payload can be found. See `pkgsend(1)`. The `hash` attribute can be used instead of the positional attribute, should the value include an equals sign. Both can be used in the same action. However, the hashes must be identical.

Other attributes include:

**preserve**

This specifies that the file's contents should not be overwritten on upgrade if the contents are determined to have changed since the file was installed or last upgraded. On initial installs, if an existing file is found, the file is salvaged (stored in `/var/pkg/lost+found`).

- If the value of `preserve` is `renameold`, then the existing file is renamed with the extension `.old`, and the new file is put in its place.
- If the value of `preserve` is `renamenew`, then the existing file is left alone, and the new file is installed with the extension `.new`.
- If the value of `preserve` is `legacy`, then this file is not installed for initial package installs. On upgrades, any existing file is renamed with the extension `.legacy`, and then the new file is put in its place.
- If the value of `preserve` is `true` (or a value not listed above, such as `strawberry`), then the existing file is left alone, and the new file is not installed. Other values with specific meanings might be added in future, so using `true` should be used if this functionality is required.

**overlay**

This specifies whether the action allows other packages to deliver a file at the same location or whether it delivers a file intended to overlay another. This functionality is intended for use with configuration files that do not participate in any self-assembly (for example, `/etc/motd`) and that can be safely overwritten.

- If `overlay` is not specified, multiple packages cannot deliver files to the same location.
- If the value of `overlay` is `allow`, one other package is allowed to deliver a file to the same location. This value has no effect unless the `preserve` attribute is also set.
- If the value of `overlay` is `true`, the file delivered by the action overwrites any other action that has specified `allow`.

  Changes to the installed file are preserved based on the value of the `preserve` attribute of the overlaying file. On removal, the contents of the file are preserved if the action being overlaid is still installed, regardless of whether the `preserve` attribute was specified. Only one action can overlay another, and the `mode`, `owner`, and `group` attributes must match.

**original_name**

This attribute is used to handle editable files moving from package to package or from place to place, or both. The form this takes is the name of the originating package, followed by a colon and the original path to the file. Any file being deleted is recorded either with its package and path, or with the value of the `original_name` attribute if specified. Any editable file being installed that has the `original_name` attribute set uses the file of that name if it is deleted as part of the same packaging operation.

Note that once set, this attribute should never change even if the package or file are repeatedly renamed; this will permit upgrade to occur from all previous versions.

**revert-tag**

This attribute is used to tag editable files that should be reverted as a set. Multiple `revert-tag` values can be specified The file reverts to its manifest-defined state when `pkg revert` is invoked with any of those tags specified. See `pkg(1)`.

Specific types of file can have additional attributes. For ELF files, the following attributes are recognized:

**elfarch**

The architecture of the ELF file. This will is the output of `uname -p` on the architecture for which the file is built.

**elfbits**

This is `32` or `64`.

**elfhash**

This is the hash of the 'interesting' ELF sections in the file. These are the sections that are mapped into memory when the binary is loaded.

These are the only sections necessary to consider when determining whether the executable behavior of two binaries will differ.

An example `file` action is:

```
file path=usr/bin/pkg owner=root group=bin mode=0755
```

### Directory Actions

The `dir` action is like the `file` action in that it represents a file system object, except that it represents a directory instead of an ordinary file. The `dir` action has the same four standard attributes as the `file` action (`path`, `owner`, `group` and `mode`), and `path` is the key attribute.

Directories are reference counted in IPS. When the last package that either explicitly or implicitly references a directory no longer does so, that directory is removed. If that directory contains unpackaged file system objects, those items are moved into `/var/pkg/lost+found`.

To move unpackaged contents into a new directory, the following attribute might be useful:

**salvage-from**

This names a directory of salvaged items. A directory with such an attribute inherits on creation the salvaged directory contents if they exist.

During installation, `pkg(1)` will check that all instances of a given directory action on the system have the same owner, group and mode attributes, and will not install the action if conflicting actions will exist on the system as a result of the operation.

An example of a `dir` action is:

```
dir  path=usr/share/lib owner=root group=sys mode=0755
```

### Link Actions

The `link` action represents a symbolic link. The `link` action has the following standard attributes:

**path**

The file system path where the symbolic link is installed. This is a `link` action's key attribute.

**target**

The target of the symbolic link. The file system object to which the link resolves.

The `link` action also takes attributes that allow for multiple versions or implementations of a given piece of software to be installed on the system at the same time. Such links are *mediated*, and allow administrators to easily toggle which links point to which version or implementation as desired. These *mediated links* are discussed in Chapter 10.

An example of a `link` action is:

```
link path=usr/lib/libpython2.6.so target=libpython2.6.so.1.0
```

### Hardlink Actions

The `hardlink` action represents a hard link. It has the same attributes as the link action, and `path` is also its key attribute.

An example of a `hardlink` action is:

```
hardlink path=opt/myapplication/hardlink target=foo
```

### Set Actions

The `set` action represents a package-level attribute, or metadata, such as the package description.

The following attributes are recognized:

**name**

The name of the attribute.

**value**

The value given to the attribute.

The `set` action can deliver any metadata the package author chooses. However, there are a number of well-defined attribute names that have specific meaning to the packaging system.

**pkg.fmri**

The name and version of the containing package.

**info.classification**

One or more tokens that a `pkg(5)` client can use to classify the package. The value should have a scheme (such as `org.opensolaris.category.2008` or `org.acm.class.1998`) and the actual classification, such as `Applications/Games`, separated by a colon (‘`:`’). The scheme is used by the `packagemanager(1)` GUI. A set of `info.classification` values is included in Appendix A.

**pkg.summary**

A brief synopsis of the description. This is output with `pkg list -s` at the end of each line, as well as in one line of the output of `pkg info`, so it should be no longer than sixty characters. It should describe *what* a package is, and should refrain from repeating the name or version of the package.

**pkg.description**

A detailed description of the contents and functionality of the package, typically a paragraph or so in length. It should describe *why* someone might want to install the package.

**pkg.obsolete**

When `true`, the package is marked obsolete. An obsolete package can have no actions other than more `set` actions, and must not be marked renamed. Package obsoletion is covered in Chapter 10

**pkg.renamed**

When `true`, the package has been renamed. There must be one or more `depend` actions in the package as well which point to the package versions to which this package has been renamed. A package cannot be marked both renamed and obsolete, but otherwise can have any number of `set` actions. Package renaming is covered in Chapter 10.

**pkg.human-version**

The version scheme used by IPS is strict, and does not allow for letters or words in the `pkg.fmri` version field. If there is a commonly used human-readable version available for a given package, that can be set here, and is displayed by IPS tools. It does not get used as a basis for version comparison and cannot be used in place of the `pkg.fmri` version.

Some additional informational attributes, as well as some used by OpenIndiana are described in Chapter 13.

An example of a `set` action is:

```
set name=pkg.summary value="Image Packaging System"
```

### Driver Actions

The driver action represents a device driver. The driver action does not reference a payload. The driver files themselves must be installed as `file` actions. The following attributes are recognized (see `add_drv(1M)` for more information):

**name**

The name of the driver. This is usually, but not always, the file name of the driver binary. This is the `driver` action's key attribute.

**alias**

This represents an alias for the driver. A given driver can have more than one `alias` attribute. No special quoting rules are necessary.

**class**

This represents a driver class. A given driver can have more than one `class` attribute.

**perms**

This represents the file system permissions for the driver's device nodes.

**clone_perms**

This represents the file system permissions for the clone driver's minor nodes for this driver.

**policy**

This specifies additional security policy for the device. A given driver can have more than one `policy` attribute, but no minor device specification can be present in more than one attribute.

**privs**

This specifies privileges used by the driver. A given driver can have more than one `privs` attribute.

**devlink**

This specifies an entry in `/etc/devlink.tab`. The value is the exact line to go into the file, with tabs denoted by '`\t`'. See `devlinks(1M)` for more information. A given driver can have more than one `devlink` attribute.

An example of a driver action is:

```
driver name=vgatext \
    alias=pciclass,000100 \
    alias=pciclass,030000 \
    alias=pciclass,030001 \
    alias=pnpPNP,900 variant.arch=i386 variant.opensolaris.zone=global
```

### Depend Actions

The `depend` action represents an inter-package dependency. A package can depend on another package because the first requires functionality in the second for the functionality in the first to work, or even to install. Dependencies are covered in more detail in Chapter 6.

The following attributes are recognized:

**fmri**

The FMRI representing the target of the dependency. This is the dependency action's key attribute. The FMRI value must not include the publisher. The package name is assumed to be complete (that is, rooted), even if it does not begin with a forward slash ('`/`'). Dependencies of type `require-any` can

have multiple `fmri` attributes. A version is optional on the `fmri` value, though for some types of dependencies, an FMRI with no version has no meaning.

The FMRI value cannot use asterisks, and cannot use the `latest` token for a version.

**type**

The type of the dependency.

- If the value is `require`, then the target package is required and must have a version equal to or greater than the version specified in the `fmri` attribute. If the version is not specified, any version satisfies the dependency. A package cannot be installed if any of its required dependencies cannot be satisfied.
- If the value is `optional`, then the target, if present, must be at the specified version level or greater.
- If the value is `exclude`, then the containing package cannot be installed if the target is present at the specified version level or greater. If no version is specified, the target package cannot be installed concurrently with the package specifying the dependency.
- If the value is `incorporate`, then the dependency is optional, but the version of the target package is constrained. See Chapter 6 for a discussion of constraints and freezing.
- If the value is `require-any`, then any one of multiple target packages as specified by multiple `fmri` attributes can satisfy the dependency, following the same rules as the `require` dependency type.
- If the value is `conditional`, the target is required only if the package defined by the `predicate` attribute is present on the system.
- If the value is `origin`, the target must, if present, be at the specified value or better on the image to be modified prior to installation. If the value of the `root-image` attribute is `true`, the target must be present on the image rooted at '`/`' in order to install this package.
- If the value is `group`, the target is required unless the package is on the image avoid list. Note that obsolete packages silently satisfy the `group` dependency. See the `avoid` subcommand in the `pkg(1)` man page.
- If the value is `parent`, then the dependency is ignored if the image is not a child image, such as a zone. If the image is a child image then the target is required to be present in the parent image. The version matching for a `parent` dependency is the same as that used for `incorporate` dependencies.

**predicate**

The FMRI representing the predicate for `conditional` dependencies.

**root-image**

Has an effect only for `origin` dependencies as mentioned above.

An example of a `depend` action is:

```
depend fmri=crypto/ca-certificates type=require
```

### License Actions

The `license` action represents a license or other informational file associated with the package contents. A package can deliver licenses, disclaimers, or other guidance to the package installer through the use of the license action.

The payload of the license action is delivered into the image metadata directory related to the package, and should only contain human-readable text data. It should not contain HTML or any other form of markup. Through attributes, license actions can indicate to clients that the related payload must be displayed and/or require acceptance of it. The method of display and/or acceptance is at the discretion of clients.

The following attributes are recognized:

**license**

This attribute provides a meaningful description for the license to assist users in determining the contents without reading the license text itself. Some example values include:

- ABC Co. Copyright Notice
- ABC Co. Custom License
- Common Development and Distribution License 1.0 (CDDL)
- GNU General Public License 2.0 (GPL)
- GNU General Public License 2.0 (GPL) Only
- MIT License
- Mozilla Public License 1.1 (MPL)
- Simplified BSD License

Wherever possible, including the version of the license in the description is recommended as shown above. The license value must be unique within a package.

**must-accept**

When `true`, this license must be accepted by a user before the related package can be installed or updated. Omission of this attribute is equivalent to `false`. The method of acceptance (interactive or configuration-based, for example) is at the discretion of clients.

**must-display**

When `true`, the action's payload must be displayed by clients during packaging operations. Omission of this value is considered equivalent to `false`. This attribute should not be used for copyright notices, only actual licenses or other material that must be displayed during operations. The method of display is at the discretion of clients.

The `license` attribute is the key attribute for the license action.

An example of a `license` action is:

```
license license="Apache v2.0"
```

### Legacy Actions

The `legacy` action represents package data used by the legacy SVR4 packaging system. The attributes associated with this action are added into the legacy system's databases so that the tools querying those databases can operate as if the legacy package were actually installed. In particular, this should be sufficient to convince the legacy system that the package named by the `pkg` attribute is installed on the system, so that the package can be used to satisfy SVR4 dependencies.

The following attributes, named in accordance with the parameters in `pkginfo(4)`, are recognized:

**category**

The value for the CATEGORY parameter. The default value is `system`.

**desc**

The value for the DESC parameter.

**hotline**

The value for the HOTLINE parameter.

**name**

The value for the NAME parameter. The default value is '`none provided`'.

**pkg**

The abbreviation for the package being installed. The default value is the name from the FMRI of the package.

**vendor**

The value for the VENDOR parameter.

**version**

The value for the VERSION parameter. The default value is the version from the FMRI of the package.

The `pkg` attribute is the key attribute for the legacy action.

An example of a `legacy` action is:

```
legacy pkg=SUNWcsu arch=i386 category=system \
    desc="core software for a specific instruction-set architecture" \
    hotline="Please contact your local service provider" \
    name="Core Solaris, (Usr)" vendor=illumos \
    version=11.11,REV=2009.11.11
```

### Signature Actions

Signature actions are used as part of the support for package signing in IPS. They are covered in detail in Chapter 11.

### User Actions

The user action defines a UNIX user as defined in `/etc/passwd`, `/etc/shadow`, `/etc/group`, and `/etc/ftpd/ftpusers` files. Users defined with this action have entries added to the appropriate files.

The following attributes are recognized:

**username**

The unique name of the user.

**password**

The encrypted password of the user. The default value is '`*LK*`'.

**uid**

The unique numeric ID of the user. The default value is the first free value under 100.

**group**

The name of the user's primary group. This must be found in `/etc/group`.

**gcos-field**

The real name of the user, as represented in the GECOS field in `/etc/passwd`. The default value is the value of the `username` attribute.

**home-dir**

The user's home directory. The default value is '`/`'.

**login-shell**

The user's default shell. The default value is empty.

**group-list**

Secondary groups to which the user belongs. See `group(4)`.

**ftpuser**

Can be set to `true` or `false`. The default value of `true` indicates that the user is permitted to login via FTP. See `ftpusers(4)`.

**lastchg**

The number of days between January 1, 1970, and the date that the password was last modified. The default value is empty.

**min**

The minimum number of days required between password changes. This field must be set to 0 or above to enable password aging. The default value is empty.

**max**

The maximum number of days the password is valid. The default value is empty. See `shadow(4)`.

**warn**

The number of days before password expires that the user is warned.

**inactive**

The number of days of inactivity allowed for that user. This is counted on a per-machine basis. The information about the last login is taken from the machine's lastlog file.

**expire**

An absolute date expressed as the number of days since the UNIX Epoch (January 1, 1970). When this number is reached, the login can no longer be used. For example, an `expire` value of 13514 specifies a login expiration of January 1, 2007.

**flag**

Set to empty.

For more information on the values of these attributes, see the `shadow(4)` man page.

A example of a user action is:

```
user gcos-field="pkg(5) server UID" group=pkg5srv uid=97 username=pkg5srv
```

### Group Actions

The group action defines a UNIX group as defined in `group(4)`. No support is present for group passwords. Groups defined with this action initially have no user list. Users can be added with the user action. The following attributes are recognized:

**groupname**

The value for the name of the group.

**gid**

The group's unique numeric id. The default value is the first free group under 100.

An example of a group action is:

```
group groupname=pkg5srv gid=97
```

## Repository

A software repository contains packages for one or more publishers. Repositories can be configured for access in a variety of different ways: HTTP, HTTPS, file (on local storage or via NFS or SMB) and as a self-contained package archive file, usually with the `.p5p` extension.

Package archives allow for convenient distribution of IPS packages, and are discussed further in Chapter 4.

A repository accessed via HTTP or HTTPS has a server process (`pkg.depotd(1M)`) associated with it; in the case of file repositories, the repository software runs as part of the accessing client.

Repositories are created with the `pkgrepo(1)` command, and package archives are created with the `pkgrecv(1)` command.

# Chapter 4

## Packaging Software with IPS

This chapter describes how to package your software with IPS.

Packaging software with IPS is usually straightforward due to amount of automation that is provided. Automation avoids repetitive tedium since that seems to be the principle cause of most packaging bugs.

Publication in IPS consists of the following steps:

1. Generate a package manifest.
2. Add necessary metadata to the generated manifest.
3. Evaluate dependencies.
4. Add any facets or actuators that are needed.
5. Verify the package.
6. Publish the package.
7. Test the package.

Each step is covered in the following sections.

### Generate a Package Manifest

The easiest way to get started is to organize the component files into the same directory structure that you want on the installed system.

This can be done with `install` target in Makefiles, or if the software you want to package is already in a tarball, unpacking the tarball into a subdirectory. For many open source software packages that use `autoconf(1)`, setting the DESTDIR environment variable to point to the desired prototype area accomplishes this.

Suppose your software consists of a binary, a library and a man page, and you want to install this software in a directory under `/opt` named `mysoftware`. You should create a directory (named `proto` in the examples) in your build area under which your software appears; e.g:

```
proto/opt/mysoftware/lib/mylib.so.1
proto/opt/mysoftware/bin/mycmd
proto/opt/mysoftware/man/man1/mycmd.1
```

Now, let's generate a manifest for this proto area. We pipe it through `pkgfmt(1)` to format the manifest so that is more readable. Assuming that the `proto` directory is in the current working directory:

```
$ pkgsend generate proto | pkgfmt > mypkg.p5m.1
```

Examining the file, you will see it contains the following lines:

```
dir path=opt group=bin mode=0755 owner=root
dir path=opt/mysoftware group=bin mode=0755 owner=root
dir path=opt/mysoftware/bin group=bin mode=0755 owner=root
dir path=opt/mysoftware/lib group=bin mode=0755 owner=root
dir path=opt/mysoftware/man group=bin mode=0755 owner=root
dir path=opt/mysoftware/man/man1 group=bin mode=0755 owner=root
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd group=bin \
    mode=0755 owner=root
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    group=bin mode=0644 owner=root
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    group=bin mode=0644 owner=root
```

The path of the files to be packaged appears twice in the file action:

- The first word after the word '`file`' describes the location of the file in the proto area.
- The path in the '`path=`' attribute specifies the location where the file is to be installed.

This double entry enables you to modify the installation location without modifying the `proto` area. This capability can save significant time, for example if you repackage software that was designed for installation on a different operating system.

Also, note that `pkgsend generate` has applied defaults for directory owners and groups. In the case of `/opt`, the defaults are not correct; we'll just delete that directory, since it's delivered by other packages already on the system and `pkg(1)` would not install the package if the attributes of `/opt` conflicted with those already on the system.

## Add Necessary Metadata to the Generated Manifest

A package should define the following metadata. See also "Set Actions" in Chapter 3.

- `pkg.fmri` defines the name and version of the package as described in "Package" in Chapter 3. A description of OpenIndiana versioning can be found in Chapter 13
- `pkg.description` is a description of the contents of the package.
- `pkg.summary` is a one-line synopsis of the description.
- `variant.arch` enumerates the architectures for which this package is suitable. If the entire package can be installed on any architecture, this can be omitted. Producing packages that have different components for different architectures is discussed in Chapter 7.
- `info.classification` is a grouping scheme used by `packagemanager(1)`, the IPS GUI. The supported values are shown in Appendix A. In this case, we pick an arbitrary one for our sample package.

In addition, we will add a link action to `/usr/share/man/index.d` pointing to our `man` directory, and discuss this link when covering *facets* and *actuators* later in this chapter.

Rather than modifying the generated manifest directly, we'll use `pkgmogrify(1)` to edit the generated manifest. A full description of how `pkgmogrify(1)` can be used to modify package manifests can be found in Chapter 8.

In this example the macro capability is used to define the architecture, as well as regular expression matching for the directory to elide from the manifest.

Now we create a small file containing the information we want to add to the manifest, as well as the transform needed to drop the `opt` directory from the manifest:

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.summary value="This is our example package"
set name=pkg.description value="This is a full description of \
all the interesting attributes of this example package."
set name=variant.arch value=$(ARCH)
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
<transform dir path=opt$->drop>
```

Running `pkgmogrify(1)` over `mypkg.p5m.1` with the above lines in a file named mypkg.mog:

```
$ pkgmogrify -DARCH=`uname -p` mypkg.p5m.1 mypkg.mog  | pkgfmt > mypkg.p5m.2
```

Examining the file we see:

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this example package. "
set name=pkg.summary value="This is our example package"
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
dir path=opt/mysoftware group=bin mode=0755 owner=root
dir path=opt/mysoftware/bin group=bin mode=0755 owner=root
dir path=opt/mysoftware/lib group=bin mode=0755 owner=root
dir path=opt/mysoftware/man group=bin mode=0755 owner=root
dir path=opt/mysoftware/man/man1 group=bin mode=0755 owner=root
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd group=bin \
    mode=0755 owner=root
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    group=bin mode=0644 owner=root
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    group=bin mode=0644 owner=root
link path=usr/share/man/index.d/mysoftware target=../../../../opt/mysoftware/man
```

Note that the directory action defining `opt` has been removed, and the manifest contents from `mypkg.mog` have been added to our package.

## Evaluate Dependencies

Use the `pkgdepend(1)` command to automatically generate dependencies for the package. The generated depend actions are defined in Chapter 3 and discussed further in Chapter 6.

Dependency generation is composed of two separate steps:

1. Determine the files on which our software depends.
2. Determine the packages that contain those files.

These steps are referred to as *dependency generation* and *dependency resolution* and are performed using the `generate` and `resolve` subcommands of `pkgdepend(1)`, respectively.

First, we'll generate our dependencies:

```
$ pkgdepend generate -md proto mypkg.p5m.2 | pkgfmt > mypkg.p5m.3
```

The `-m` option causes `pkgdepend(1)` to include the entire manifest in its output, and the `-d` option passes the `proto` directory to the command.

In this new file, we see:

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this example package."
set name=pkg.summary value="This is our example package"
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir path=opt/mysoftware group=bin mode=0755 owner=root
dir path=opt/mysoftware/bin group=bin mode=0755 owner=root
dir path=opt/mysoftware/lib group=bin mode=0755 owner=root
dir path=opt/mysoftware/man group=bin mode=0755 owner=root
dir path=opt/mysoftware/man/man1 group=bin mode=0755 owner=root
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd group=bin \
    mode=0755 owner=root
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    group=bin mode=0644 owner=root
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    group=bin mode=0644 owner=root
link path=usr/share/man/index.d/mysoftware target=../../../../opt/mysoftware/man
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/bin/mycmd \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=opt/mysoftware/lib pkg.debug.depend.path=usr/lib
depend fmri=__TBD pkg.debug.depend.file=libc.so.1 \
    pkg.debug.depend.reason=opt/mysoftware/lib/mylib.so.1 \
    pkg.debug.depend.type=elf type=require pkg.debug.depend.path=lib \
    pkg.debug.depend.path=usr/lib
```

`pkgdepend(1)` has added notations about a dependency on `libc.so.1` by both `mylib.so.1` and `mycmd`. Note that the internal dependency between `mycmd` and `mylib.so.1` is currently silently elided by `pkgdepend(1)`.

Now we need to resolve these dependencies. To resolve dependencies, `pkgdepend(1)` examines the packages currently installed on the machine used for building the software. By default, `pkgdepend(1)` puts its output in `mypkg.p5m.3.res`. Note that this takes a while to run as it loads lots of information about the system on which it is running. `pkgdepend(1)` will resolve many packages at once if you want to amortize this time over all packages; running it on one package at a time is not time efficient.

```
$ pkgdepend resolve -m mypkg.p5m.3
```

When this completes, `mypkg.p5m.3.res` contains:

```
set name=pkg.fmri value=mypkg@1.0,5.11-0
set name=pkg.description \
    value="This is a full description of all the interesting attributes of this example package."
set name=pkg.summary value="This is our example package"
set name=info.classification \
    value=org.opensolaris.category.2008:Applications/Accessories
set name=variant.arch value=i386
dir path=opt/mysoftware group=bin mode=0755 owner=root
dir path=opt/mysoftware/bin group=bin mode=0755 owner=root
dir path=opt/mysoftware/lib group=bin mode=0755 owner=root
dir path=opt/mysoftware/man group=bin mode=0755 owner=root
dir path=opt/mysoftware/man/man1 group=bin mode=0755 owner=root
file opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd group=bin \
    mode=0755 owner=root
file opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    group=bin mode=0644 owner=root
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    group=bin mode=0644 owner=root
link path=usr/share/man/index.d/mysoftware target=opt/mysoftware/man
depend fmri=pkg:/system/library@0.5.11,5.11-0.175.0.0.0.2.1 type=require
```

`pkgdepend(1)` has converted the notation about the file dependency on `libc.so.1` to a package dependency on `pkg:/system/library` which delivers that file.

We recommended that developers use `pkgdepend(1)` to generate dependencies, rather than declaring `depend` actions manually. Manual dependencies can become incorrect or unnecessary as the package contents might change over time. This could happen, for example, when a file that an application depends on gets moved to a different package. Any manually declared dependencies on that package would then be out of date.

Some manually declared dependencies might be necessary if `pkgdepend(1)` is unable to determine dependencies completely, in which case we recommend that comments are added to the manifest to explain the nature of each dependency.

## Add Any Facets or Actuators That Are Needed

Facets and actuators are discussed in more detail in Chapter 7 and Chapter 9. Facets allow us to denote actions that are not required but can be optionally installed. Actuators specify system changes that must occur when an action in our package is installed, updated, or removed.

Since we are delivering a man page in `opt/mysoftware/man/man1` we would like to add a facet to indicate that documentation is optional.

We would also like an SMF service, `svc:/application/man-index:default`, to be restarted when our package is installed, so that our man page is included in the index. The 'restart_fmri' actuator can perform that task.

The `man-index` service looks in `/usr/share/man/index.d` for symbolic links to directories containing man pages, adding the target of each link to the list of directories it scans, hence our earlier addition of that link to our man pages. This is a good example of the *self-assembly* idiom that was discussed in Chapter 1, and is used throughout the packaging of the OS itself.

OpenIndiana ships with a set of `pkgmogrify(1)` transforms that were used to package the the operating system, in `/usr/share/pkg/transforms`. These transforms are discussed in more detail in Chapter 8.

The file called `documentation` contains the transforms that are closest to what we need here, though since we're delivering our man page to `/opt`, we'll use the `documentation` transforms file as a guide, and use the following transforms instead. These transforms include a regular expression `opt/.+/man(/.+)?` which match all paths beneath `opt` that contain a `man` subdirectory:

```
<transform dir file link hardlink path=opt/.+/man(/.+)? -> \
    default facet.doc.man true>

<transform file path=opt/.+/man(/.+)? -> \
    add restart_fmri svc:/application/man-index:default>
```

We can run our manifest through this transform using:

```
$ pkgmogrify mypkg.p5m.3.res /tmp/doc-transform | pkgfmt > mypkg.p5m.4.res
```

which changes the three man-page-related actions in our manifest, from:

```
dir path=opt/mysoftware/man group=bin mode=0755 owner=root
dir path=opt/mysoftware/man/man1 group=bin mode=0755 owner=root
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    group=bin mode=0644 owner=root
```

to:

```
dir  path=opt/mysoftware/man owner=root group=bin mode=0755 facet.doc.man=true
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755 \
    facet.doc.man=true
file opt/mysoftware/man/man1/mycmd.1 path=opt/mysoftware/man/man1/mycmd.1 \
    owner=root group=bin mode=0644 \
    restart_fmri=svc:/application/man-index:default facet.doc.man=true
```

For efficiency, we could have included this transform when originally adding metadata to our package, before running `pkgdepend(1)`.

## Verify the Package

The last thing we need to do before publication is run `pkglint(1)` on our manifest. This helps us determine whether we've made any errors while writing the manifest that we'd like to catch before publication. Some of the errors that `pkglint(1)` can catch are ones also caught either at publication time, or when a user tries to install a package, but obviously, we'd like to catch errors as early as possible in the package authoring process.

For example, `pkglint(1)` checks that the package doesn't deliver files already owned by another package, and that all metadata for shared, reference-counted actions (such as directories) is consistent across packages.

There are two modes in which to run `pkglint(1)`:

- Directly on the manifest itself
- On the manifest, also referencing a repository

For developers who want to quickly check the validity of their manifests, using the first form is usually sufficient. The second form is recommended to be run *at least once* before publication to a repository.

By referencing a repository, `pkglint(1)` can perform additional checks to ensure that the package interacts well with other packages in that repository.

The full list of checks that `pkglint(1)` performs can be shown with `pkglint -L`. Detailed information on how to enable, disable and bypass particular checks is given in the `pkglint(1)` man page. It also details how to extend `pkglint(1)` to run additional checks.

In the case of our test package, we see:

```
$ pkglint mypkg.p5m.4.res
Lint engine setup...
Starting lint run...
WARNING opensolaris.manifest001.1 Missing attribute
    'org.opensolaris.consolidation' in pkg:/mypkg@1.0,5.11-0
WARNING pkglint.action005.1       obsolete dependency check skipped: unable
    to find dependency pkg:/system/library@0.5.11-0.168 for
    pkg:/mypkg@1.0,5.11-0
```

These warnings are acceptable for our purposes:

- `opensolaris.manifest001.1` is warning us that we haven't declared a tag that is generally only required for bundled OpenIndiana software, so we can ignore this warning.
- `pkglint.action005.1` is warning us that `pkglint(1)` wasn't able to find a package called `pkg:/system/library@0.5.11-0.168` which we have generated a dependency on. Since `pkglint(1)` was called with just the manifest file as an argument, it does not know which repository that package is present in, hence the warning.

When `pkglint(1)` is run with a `-r` flag referencing a repository containing the package that our test package has a dependency on, we see:

```
$ pkglint -c ./oi-reference -r http://pkg.openindiana.org/hipster mypkg.p5m.4.res
Lint engine setup...

PHASE                                        ITEMS
4                                         4292/4292
Starting lint run...

WARNING opensolaris.manifest001.1 Missing attribute 'org.opensolaris.consolidation' in pkg:/mypkg@1.0,5.11-0
$
```

## Publish the Package

Now that our package is created, dependencies are added, and it has been checked for correctness, we can publish the package.

IPS provides three different ways to deliver a package:

- Publish to a local file-based repository
- Publish to a remote HTTP-based repository
- Convert to a `.p5p` package archive

Generally, publishing to a file-based repository is sufficient while testing a package.

If the package needs to be transferred to other machines which cannot access the package repositories, converting one or more packages to a package archive can be convenient.

The package can also be published directly to an HTTP repository, hosted on a machine with a read/write instance of `svc:/application/pkg/server` (which in turn runs `pkg.depotd(1M)`).

We do not generally recommend this method of publication since there are no authorization/authentication checks on the incoming package when publishing over HTTP. Publishing to HTTP repositories can be convenient on secure networks or when testing the same package across several machines if NFS or SMB access to the file repository is not possible.

Installing packages over HTTP (or preferably HTTPS) is fine, however.

### Local File Repositories

`pkgrepo(1)` can be used to create and manage repositories. We choose a location on our system, create a repository, then set the default publisher for that repository:

```
$ pkgrepo create /scratch/my-repository
$ pkgrepo -s /scratch/my-repository set publisher/prefix=mypublisher
$ find /scratch/my-repository/
/scratch/my-repository/
/scratch/my-repository/pkg5.repository
```

We can now use `pkgsend` to publish our package, and `pkgrepo` to examine the repository afterwards:

```
$ pkgsend -s /scratch/my-repository/ publish -d proto mypkg.p5m.4.res
pkg://mypublisher/mypkg@1.0,5.11-0:20111012T034303Z
PUBLISHED
$ pkgrepo -s /scratch/my-repository info
PUBLISHER   PACKAGES STATUS           UPDATED
mypublisher 1        online           2011-10-12T03:43:04.117536Z
```

The file repository can then be served over HTTP or HTTPS using `pkg.depotd(1M)` if required.

### Package Archives

Package archives enable you to distribute groups of packages in a single file. We can use `pkgrecv(1)` to create package archives from package repositories, and vice versa.

Package archives can be easily downloaded from an existing website, copied to a USB key or burned to a DVD for installation in cases where a package repository is not available.

In the case of our simple file repository above, we can create an archive from this repository with the following command:

```
$ pkgrecv -s /scratch/my-repository -a -d myarchive.p5p mypkg
Retrieving packages for publisher mypublisher ...
Retrieving and evaluating 1 package(s)...
DOWNLOAD                                  PKGS       FILES    XFER (MB)
Completed                                  1/1         3/3      0.7/0.7


ARCHIVE                                               FILES   STORE (MB)
myarchive.p5p                                         14/14      0.7/0.7
```

We can list the newest available packages from a repository using pkgrepo:

```
$ pkgrepo -s /scratch/my-repository list '*@latest'
PUBLISHER    NAME                            O VERSION
mypublisher  mypkg                             1.0,5.11-0:20111012T033207Z
```

This output can be useful when constructing scripts to create archives with the latest versions of all packages from a given repository.

Temporary repositories or package archives provided with the `-g` flag for `pkg install` and other package operations cannot be used on systems with child or parent images (non-global zones have a child/parent relationship with the global zone) since the system repository does not get temporarily configured with that publisher information.

Package archives can be set as sources of local publishers in non-global zones, however.

## Test the Package

Having published our package, we are interested in seeing whether it has been packaged properly.

In this example, we ensure that our user has the *Software Installation* Profile, in order to be able to install packages without root privileges, then we add the publisher in our repository to the system:

```
$ sudo su
Password:
# usermod -P 'Software Installation' myuser
Found user in files repository.
UX: usermod: myuser is currently logged in, some changes may not take effect
until next login.
^D
$ pfexec pkg set-publisher -p /scratch/my-repository
pkg set-publisher:
  Added publisher(s): mypublisher
```

You can use `pkg install` -nv to see what the install command will do without making any changes. The following example actually installs the package:

```
$ pfexec pkg install mypkg
           Packages to install:  1
       Create boot environment: No
Create backup boot environment: No

DOWNLOAD                                  PKGS       FILES    XFER (MB)
Completed                                  1/1         3/3      0.7/0.7

PHASE                                       ACTIONS
Install Phase                                 15/15

PHASE                                         ITEMS
Package State Update Phase                      1/1
Image State Update Phase                        2/2

PHASE                                         ITEMS
Reading Existing Index                          8/8
Indexing Packages                               1/1
```

We can then examine the software as it was delivered on the system:

```
$ find /opt/mysoftware/
/opt/mysoftware/
/opt/mysoftware/bin
/opt/mysoftware/bin/mycmd
/opt/mysoftware/lib
/opt/mysoftware/lib/mylib.so.1
/opt/mysoftware/man
/opt/mysoftware/man/man-index
/opt/mysoftware/man/man-index/term.doc
/opt/mysoftware/man/man-index/.index-cache
/opt/mysoftware/man/man-index/term.dic
/opt/mysoftware/man/man-index/term.req
/opt/mysoftware/man/man-index/term.pos
/opt/mysoftware/man/man1
/opt/mysoftware/man/man1/mycmd.1
```

In addition to the binaries and man page showing up, we can see that the system has also generated the man page indexes as a result of our actuator restarting the `man-index` service.

We can see that `pkg info` shows the metadata that we added to our package:

```
$ pkg info mypkg
          Name: mypkg
       Summary: This is our example package
   Description: This is a full description of all the interesting attributes of
                this example package.
      Category: Applications/Accessories
         State: Installed
     Publisher: mypublisher
       Version: 1.0
 Build Release: 5.11
        Branch: 0
Packaging Date: October 12, 2011 03:43:03 AM
          Size: 1.75 MB
          FMRI: pkg://mypublisher/mypkg@1.0,5.11-0:20111012T034303Z
```

We can also see that `pkg search` returns hits when querying for files in our package:

```
$ pkg search -l mycmd.1
INDEX      ACTION VALUE                              PACKAGE
basename   file   opt/mysoftware/man/man1/mycmd.1 pkg:/mypkg@1.0-0
```

# Chapter 5

# Installing, Removing, and Updating Software Packages

This chapter describes how the IPS client works internally when installing, updating and removing the software installed in an image.

Understanding basically how `pkg(1)` works will help administrators and developers better understand the various errors that can occur, and allow them to more quickly resolve package dependency problems.

## How package changes are performed

The following steps are executed when `pkg(1)` is invoked to modify the software installed on the machine:

- Check input for errors
- Determine the system end-state
- Run basic checks
- Run the solver
- Optimize the solver results
- Evaluate actions
- Download content
- Execute actions
- Process actuators

When operating on the global zone, during execution of the steps above, `pkg(1)` can execute operations on any non-global zones on the machine, for example to ensure that dependencies are correct between the global and non-global zones, or to download content or execute actions for non-global zones. Chapter 12 has more detail about zones.

In the following sections, we'll describe each of these steps.

### Check Input for Errors

We perform basic error checking on the options presented on the command line.

### Determine the System End State

A description of the desired end state of the system is constructed. In the case of updating all packages in the image this might be something like *"all the packages currently installed, or newer versions of them"*. In the case of package removal, it would be *"all the packages currently installed without this one"*.

IPS tries hard to determine what the user intends this end state to look like. In some cases, IPS might determine an end state that is not what the user intended, even though that end state does match what the user requested.

When troubleshooting, it is best to be as *specific* as possible. The following command is not specific:

```
# pkg update
```

If this command fails with a message such as `No updates available for this image`, then you might want to try a more specific command such as the following command:

```
# pkg update "*@latest"
```

This command defines the end state more exactly, and can produce more directed error messages.

### Run Basic Checks

The desired end state of the system is reviewed to make sure that a solution appears possible. During this basic review, `pkg(1)` checks that a plausible version exists of all dependencies, and that desired packages do not exclude each other.

If an obvious error exists, then `pkg(1)` will print an appropriate error message and exit.

### Run the Solver

The solver forms the core of the computation engine used by `pkg(5)` to determine the packages that can be installed, updated or removed, given the constraints in the image and constraints introduced by any new packages for installation.

This problem is an example of a *Boolean satisfiability problem*, and can be solved by a SAT solver.

The various possible choices for all the packages are assigned boolean variables, and all the dependencies between those packages, any required packages, etc. are cast as boolean expressions in conjunctive normal form.

The set of expressions generated is passed to MiniSAT. If MiniSAT cannot find any solution, the error handling code attempts to walk the set of installed packages and the attempted operation, and print the reasons that each possible choice was eliminated.

If the currently installed set of packages meet the requirements but no other does, `pkg(1)` will report that there is nothing to do.

As mentioned in a previous section, the error message generation and specificity is determined by the inputs to `pkg(1)`. Being as specific as possible in commands issued to `pkg(1)` will produce the most useful error messages.

If on the other hand MiniSAT finds a possible solution, we begin optimization.

### Optimize the Solver Results

The optimization phase is necessary because there is no way of describing some solutions as more desirable than others to a SAT solver.

Instead, once a solution is found, IPS adds constraints to the problem to separate less desirable choices, and to separate the current solution as well. We then repeatedly invoke MiniSAT and repeat the above operation until no more solutions are found. The last successful solution is taken as the best one.

Clearly, the difficulty of finding a solution is proportional to the number of possible solutions. Being more specific about the desired result will produce solutions more quickly.

### Evaluate Actions

Once the set of package FMRIs that best satisfy the posed problem is found, the evaluation phase begins.

In this phase, we compare the packages currently installed on the system with the end state, and compare package manifests of old and new packages to determine three lists:

- Actions that are being removed
- Actions that are being added

- Actions that are being updated

The action lists are then updated so that:

- directory and link actions are reference counted, mediated link processing is done
- hardlinks are marked for repair if their target file is updated. This is done because updating a target of a hardlink in a manner that is safe for currently executing processes breaks the hard links.
- editable files moving between packages are correctly handled so that any user edits are not lost.
- the action lists are sorted so that removals, additions and updates occur in the correct order.

All the currently installed packages are then cross-checked to make sure that no packages conflict. That is, ensuring that two packages do not attempt to deliver a file to the same location, ensuring that directory attributes for the same directory agree between packages, etc.

If conflicts exist, these are reported and `pkg(1)` exits with an error message.

Finally, the action lists are scanned to determine if any SMF services need to be restarted if this operation is performed, whether or not this change can be applied to a running system, whether the boot archive needs to be rebuilt and whether the amount of space required is available, etc.

### Download Content

If `pkg(1)` is running without the `-n` flag, processing continues to the download phase.

For each action that requires content, we download any required files by hash and cache them. This step can take some time if the amount of content to be retrieved is large.

Once downloading is complete, if the change is to be applied to a live system (image is rooted at '/') and a reboot is required, the running system is cloned and the target image is switched to the clone.

### Execute Actions

Executing actions involves actually performing the install or remove methods specific to each action type on the image.

Execution begins with all the removal actions being executed. If any unexpected content is found in directories being removed from the system, that content is placed in `/var/pkg/lost+found`.

Execution then proceeds to install and update actions. Note that all the actions have been blended across all packages. Thus all the changes in a single package operation are applied to the system at once rather than package by package. This permits packages to depend on each other, exchange content, etc. safely. For details on how files are updated, see the description of the `file` action in Chapter 3.

### Process Actuators

If we're updating a live system, any pending actuators are executed at this point. These are typically SMF service restarts and refreshes. Once these are launched, we update the local search indicies. We discuss actuators in more detail in Chapter 9

Lastly, if needed, we update the boot archive.

# Chapter 6

## Specifying Dependencies

Dependencies define how packages are related.

IPS provide a variety of different dependency types as discussed in Chapter 3. In this chapter we go into more detail about how each dependency type can be used to control the software that is installed.

In IPS, a package cannot be installed unless all package dependencies are satisfied. IPS allows packages to be mutually dependent (to have circular dependencies). IPS also allows packages to have different kinds of dependencies on the same package at the same time.

### Dependency Types

Each section below contains an example `depend` action, as it would appear in a manifest during package creation.

#### `require`

The most basic type of dependency is the `require` dependency.

If a package A@1.0 contains a `require` dependency on package B@2, it means that if A@1.0 is installed, a version of B at 2 or higher must be installed as well.

This acceptance of higher versioned packages reflects the implicit expectation of binary compatibility in newer versions of existing packages.

These dependencies are typically used to express functional dependencies such as libraries or interpreters such as Python, Perl, etc. The version portion of the specified FMRI can be omitted; it indicates that any version will suffice. The latter might not be actually true, but if other dependencies constrain the version adequately, this might save some effort.

An example `require` dependency is:

```
depend fmri=pkg:/system/library type=require
```

#### `require-any`

The `require-any` dependency is used if more than one package will satisfy a functional requirement. IPS will pick one of the packages to install if the dependency is not already satisfied.

A typical use might be to ensure that at least one version of Perl was installed on the system, for example. The versioning is handled in the same manner as for the `require` dependency.

An example `require-any` dependency is:

```
depend type=require-any fmri=pkg:/editor/gnu-emacs/gnu-emacs-gtk \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-no-x11 \
    fmri=pkg:/editor/gnu-emacs/gnu-emacs-x11
```

#### `optional`

The `optional` dependency is similar to the `require` dependency, but the specified package need not be installed. However, if it is present, it must be at the specified version or greater.

This type of dependency is typically used to handle cases where packages transfer content.

In this case, each version of the package post-transfer would contain an optional dependency on the other package's post-transfer version, so it would be impossible to install incompatible versions of the two packages.

Omitting the version on an optional dependency makes the dependency a no-op, but is permitted.

An example `optional` dependency is:

```
depend fmri=pkg:/x11/server/xorg@1.9.99 type=optional
```

### `conditional`

The `conditional` dependency is similar to the `require` dependency as well, except that a predicate attribute is present. If the package specified in the value of the predicate attribute is present on the system at the specified or greater version, the conditional dependency is treated as a `require` dependency, otherwise it is ignored.

This type of dependency is most often used to install optional extensions to a package if the requisite base packages are present on the system.

For example, an editor package that has both X11 and terminal versions might place the X11 version in a separate package, and include a conditional dependency on the X11 version from the text version with the existence of the requisite X client library package as the predicate.

An example conditional dependency is:

```
depend type=conditional fmri=library/python-2/pycurl-26 \
    predicate=runtime/python-26
```

### `group`

The `group` dependency is used to construct groups of packages.

The group dependency will ignore the version specified; any version of the named package satisfies this dependency. The named package is required unless the package has been placed on the avoid list (see `pkg(1)`), the package is rejected with `pkg install --reject`, or the package is uninstalled with `pkg uninstall`.

These three options enable administrators to 'deselect' packages that are the subject of a group dependency. IPS will remember this and not re-install the package during an update unless it becomes required by another dependency. If the new dependency is removed by another subsequent operation, then the package is uninstalled again.

A good example of how to use these dependencies is to construct packages containing group dependencies on packages that are needed for typical uses of a system.

The administrator could install all that apply and know that over subsequent updates to newer versions of the OS, the appropriate packages would be added to his system.

An example `group` dependency is:

```
depend fmri=package/pkg type=group
```

### `origin`

The `origin` dependency exists to resolve upgrade issues that require intermediate transitions. The default behavior is to specify the minimum version of a package (if installed) that must be present on the system being updated.

For example, a typical use might be a database package version 5 that supports upgrade from version 3 or greater, but not earlier versions.

In this case, version 5 would have an origin dependency on itself at version 3. Thus, if version 5 was being fresh installed, installation would proceed; but if version 1 of the package was installed, one could not upgrade directly to this version.

Thus, `pkg update database-package` would not select version 5 in this case but would pick version 3 instead as the latest possible version it could upgrade to.

The behavior of this dependency can be modified by the `root-image` attribute being set to `true`; in this case the named package must be at the specified version or greater if it is present in the running system, rather than the image being updated.

This is generally used for operating system issues such as dependencies on boot block installers.

An example `origin` dependency is:

```
depend fmri=pkg:/database/mydb@3.0 type=origin
```

### `parent`

The `parent` dependency is used for zones or other child images. In this case, the dependency is only checked in the child image, and specifies a package and version that must be present in the parent image or global zone. The version specified must match to the level of precision specified.

For example, if the `parent` dependency is on A@2.1, then any version of A beginning with 2.1. will match. This dependency is often used to require that packages are kept in sync between non-global zones and the global zone, and as a short cut a special package name `feature/package/dependency/self` is used as a synonym for the exact version of the package that contains this dependency.

This is used to keep key operating system components, such as `libc.so.1` installed in the zone synchronized with the kernel installed in the global zone. The `parent` dependency is also discussed in Chapter 12.

An example `parent` dependency is:

```
depend type=parent fmri=feature/package/dependency/self \
    variant.opensolaris.zone=nonglobal
```

### `incorporate`

The `incorporate` dependency is heavily used in OpenIndiana to ensure that compatible versions of software are installed together.

The basic mechanism is like that of an `optional` dependency, except that the version matching is that of the `parent` dependency: if this package is present, it must be at the specified version to the level specified.

How these dependencies are typically used is that many of them are placed in the same package to define a surface in the package version space that is compatible. Packages that contain such sets of incorporate dependencies are often called *incorporations*; it is typical to define such for sets of software packages that are built together and are not separately versioned, like much of the kernel.

An example `incorporate` dependency is:

```
depend type=incorporate \
    fmri=pkg:/driver/network/e1000g@0.5.11,5.11-2018.0.0.18233
```

`exclude`

The `exclude` dependency is seldom used. It allows the containing package to preclude installation with the specified package at the specified version or higher.

Note that if the version is omitted, no version of the specified package can be installed with the containing package. These constraints can be frustrating to administrators, and should be avoided where possible.

An example exclude dependency is:

```
depend fmri=pkg:/x11/server/xorg@1.10.99 type=exclude
```

## Constraints and Freezing

### Constraints

Through the careful use of the various types of `depend` actions described above, packages can define the ways in which they are allowed to be upgraded.

In general, we often desire that a group of packages installed on a system be supported and upgraded as a group: Either all packages in the group are updated, or none of the packages in the group are updated. As mentioned earlier, this is the reason for using the `incorporate` dependency in OpenIndiana.

The following three partial package manifests show the relationship between the `foo` and `bar` packages and the `myincorp` incorporation package:

```
set name=pkg.fmri value=foo@1.0
dir path=foo owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

```
set name=pkg.fmri value=bar@1.0
dir path=bar owner=root group=bin mode=0755
depend fmri=myincorp type=require
```

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=foo@1.0 type=incorporate
depend fmri=bar@1.0 type=incorporate
```

The `foo` and `bar` packages both have a `require` dependency on the `myincorp` incorporation. The `myincorp` package has `incorporate` dependencies such that `foo` and `bar` can be upgraded to **at most** version 1.0, to the level of granularity defined by the version number and, if installed, must be **at least** at version 1.0 or greater.

That is, an `incorporate` dependency on version 1.0 allows for 1.0.1, 1.0.2.1, etc. but doesn't allow version 1.1, version 2.0, version 0.9, etc. When we deliver a new incorporation package, one that has `incorporate` dependencies at a higher version, we will allow `foo` and `bar` to upgrade to those versions instead (assuming that the incorporation package is also being upgraded).

Note here that because `foo` and `bar` both have `require` dependencies on the `myincorp` package, the incorporation package must always be installed.

However, conflicting with the requirement we stated at the beginning of this section, there are some situations where we might want to relax the incorporation constraint.

Perhaps `bar` can function independently of `foo`, but we'd like `foo` to remain within the series of versions our incorporation constrains it to.

We can relax the incorporation constraints using *facets*, allowing the administrator to effectively *disable* certain `incorporate` dependencies. Facets are discussed in more detail in Chapter 7. Briefly, facets are special attributes that can be applied to actions within a package to enable authors to mark those actions as optional.

When actions are marked with facet attributes in this manner, the actions containing those facets can be enabled or disabled using the `pkg change-facet` command.

By convention, facets that optionally install `incorporate` dependencies are named `facet.version-lock.<name>`, where *name* is the package name containing that `depend` action.

So, for example, using the example above, we could have the following incorporation:

```
set name=pkg.fmri value=myincorp@1.0
depend fmri=foo@1.0 type=incorporate
depend fmri=bar@1.0 type=incorporate facet.version-lock.bar=true
```

This incorporation includes our `depend` action by default, constraining `bar` to version 1.0. The following command relaxes this constraint:

```
pkg change-facet version-lock.bar=false
```

The `bar` package is free from the incorporation constraints, and can be upgraded to version 2.0 if necessary.

### Freezing

So far, all of the discussion has been around constraints that have been applied during the package authoring process, by modifying the package manifests themselves.

`pkg(1)` also has a means for the administrator to apply constraints to the system at runtime.

Using the `pkg freeze` command, the administrator can prevent a given package from being updated past either its current version, or a version specified on the command line. This capability is effectively the same as an `incorporate` dependency.

See the `pkg(1)` man page for more information on the `freeze` command.

In order to apply more complex dependencies to an image, it is necessary to create and install a package that includes those dependencies.

# Chapter 7

## Allowing Variations

In this chapter we explore how variants and facets are used in IPS to provide different installation options to the end user.

### Variants

Since OpenIndiana supports multiple architectures, one common error made with the SVR4 packaging system was the accidental installation of packages for an incorrect architecture. With the introduction of software repositories, the prospect of maintaining a separate repository for each supported architecture seemed unappealing to ISVs and error prone for customers. As a result, IPS supports installation of a single package on multiple architectures.

The mechanism that implements this feature is called a *variant*. A variant enables the properties of the target image to determine which software components are actually installed.

A variant has two parts: its name, and the list of possible values. The variants defined in OpenIndiana are:

| Name | Values |
|---|---|
| variant.arch | sparc, i386 |
| variant.opensolaris.zone | global, nonglobal |
| variant.debug.* | true, false |

Variants appear in two places in a package:

- A `set` action names the variant and defines the values that apply to this package.
- Any action that can only be installed for a subset of the variant values has a tag that specifies the name of the variant and the value on which it is installed.

For example, a package that delivers the symbolic link `/var/ld/64` might include:

```
set name=variant.arch value=sparc value=i386
dir group=bin mode=0755 owner=root path=var/ld
dir group=bin mode=0755 owner=root path=var/ld/amd64 \
    variant.arch=i386
dir group=bin mode=0755 owner=root path=var/ld/sparcv9 \
    variant.arch=sparc
link path=var/ld/32 target=.
link path=var/ld/64 target=sparcv9 variant.arch=sparc
link path=var/ld/64 target=amd64 variant.arch=i386
```

Note that components that are delivered on both sparc and i386 receive no variant tag, but those delivered to one architecture or the other receive the appropriate tag. It is perfectly reasonable for actions to contain multiple tags for different variant names; there might be debug and nondebug binaries for both sparc and i386.

In OpenIndiana, kernel components are commonly elided from packages installed in zones, as they serve no useful purpose. Thus, they are marked with the `opensolaris.zone` variant set to `global` so that they are not installed in non-global zones. This is typically done in the manifest during publication with a `pkgmogrify(1)` rule. Thus the packages from the i386 and sparc builds are already marked for zones. We then use the `pkgmerge(1)` command to take the packages from the sparc and i386 builds and merge them together. This is far more reliable and faster than attempting to construct such packages manually.

In general, it is not practical to define new variants without modifying the packaging system as no practical means currently exists for defining a default value for variants in general. Package developers cannot define new variants at present.

However, the `variant.debug.*` portion of the variant namespace is predefined to have a default version of `false`; thus, developers can provide debug versions of their components, tagged with the appropriate variant, and users can select that variant if problems arise. Remember that variants are set per image, so selecting a suitable name that is unique at the appropriate resolution for that piece of software is important.

Note that variant tags are applied to any actions that differ between architectures during merging; this includes dependencies, `set` actions, etc. Packages that are marked as not supporting one of the variant values of the current image are not considered for installation.

The `pkgmerge(1)` man page provides several examples of merging packages. Note that it will merge across multiple different variants at the same time if needed.

## Facets

Often, package developers have optional portions of their software that actually belong with the main body, but some people might not want to install. Some examples are localization files for different locales, man pages and other documentation, header files needed only by developers or DTrace users.

Traditionally, such optional content has been placed in separate packages with an arbitrarily selected naming convention (such as appending `-dev` or `-devel` to the package name) enabling administrators to select the optional content.

This has led to various problems, such as adding a new locale for all the software on a system being a rather irritating task, as the admin has to discover all the necessary packages by examining the lists of available packages.

IPS has implemented a mechanism similar to variants called *facets* to deal with this problem. Like variants, facets have a name and a value. The value is either set to `true` or `false` in the image. The default value is `true`. The facet namespace is hierarchal, with matching rules such that the longest match wins.

For example, the default value for all facets is `true`; the `pkg(1)` client implicitly sets `facet.*` to `true`. Documentation in OpenIndiana packages is tagged with the type of documentation. For example, man pages are tagged with `facet.doc.man=true` in the package manifests.

The following commands include man pages but exclude all other documentation from being installed in this image:

```
# pkg change-facet facet.doc.*=false
# pkg change-facet facet.doc.man=true
```

Similarly, the following commands install only the German localization in this image:

```
# pkg change-facet facet.locale.*=false
# pkg change-facet facet.locale.de=true
```

If an action contains multiple facet tags, the action is installed if the value of any of the facet tags is `true`.

The `pkg facet` command is useful in determining which facets are set in the image.

The package developer can use `pkgmogrify(1)` to quickly tag his man pages, localizations, etc. using regular expressions to match the different types of files. This is described in detail in Chapter 8.

Facets can also be used to manage dependencies, essentially turning dependencies on and off, depending on whether the facet is set. See Chapter 6 for a discussion of `facet.version-lock.*`.

OpenIndiana facets that might be of use for software developers include:

| | | |
|---|---|---|
| facet.devel | facet.locale.es_BO | facet.locale.lt_LT |
| facet.doc | facet.locale.es_CL | facet.locale.lv |
| facet.doc.man | facet.locale.es_CO | facet.locale.lv_LV |
| facet.doc.pdf | facet.locale.es_CR | facet.locale.mk |
| facet.doc.info | facet.locale.es_DO | facet.locale.mk_MK |
| facet.doc.html | facet.locale.es_EC | facet.locale.ml |
| facet.locale | facet.locale.es_ES | facet.locale.ml_IN |
| facet.locale.af | facet.locale.es_GT | facet.locale.mr |
| facet.locale.af_ZA | facet.locale.es_HN | facet.locale.mr_IN |
| facet.locale.ar | facet.locale.es_MX | facet.locale.ms |
| facet.locale.ar_AE | facet.locale.es_NI | facet.locale.ms_MY |
| facet.locale.ar_BH | facet.locale.es_PA | facet.locale.mt |
| facet.locale.ar_DZ | facet.locale.es_PE | facet.locale.mt_MT |
| facet.locale.ar_EG | facet.locale.es_PR | facet.locale.nb |
| facet.locale.ar_IQ | facet.locale.es_PY | facet.locale.nb_NO |
| facet.locale.ar_JO | facet.locale.es_SV | facet.locale.nl |
| facet.locale.ar_KW | facet.locale.es_US | facet.locale.nl_BE |
| facet.locale.ar_LY | facet.locale.es_UY | facet.locale.nl_NL |
| facet.locale.ar_MA | facet.locale.es_VE | facet.locale.nn |
| facet.locale.ar_OM | facet.locale.et | facet.locale.nn_NO |
| facet.locale.ar_QA | facet.locale.et_EE | facet.locale.no |
| facet.locale.ar_SA | facet.locale.eu | facet.locale.or |
| facet.locale.ar_TN | facet.locale.fi | facet.locale.or_IN |
| facet.locale.ar_YE | facet.locale.fi_FI | facet.locale.pa |
| facet.locale.as | facet.locale.fr | facet.locale.pa_IN |
| facet.locale.as_IN | facet.locale.fr_BE | facet.locale.pl |
| facet.locale.az | facet.locale.fr_CA | facet.locale.pl_PL |
| facet.locale.az_AZ | facet.locale.fr_CH | facet.locale.pt |
| facet.locale.be | facet.locale.fr_FR | facet.locale.pt_BR |
| facet.locale.be_BY | facet.locale.fr_LU | facet.locale.pt_PT |
| facet.locale.bg | facet.locale.ga | facet.locale.ro |
| facet.locale.bg_BG | facet.locale.gl | facet.locale.ro_RO |
| facet.locale.bn | facet.locale.gu | facet.locale.ru |
| facet.locale.bn_IN | facet.locale.gu_IN | facet.locale.ru_RU |
| facet.locale.bs | facet.locale.he | facet.locale.ru_UA |

| | | |
|---|---|---|
| facet.locale.bs_BA | facet.locale.he_IL | facet.locale.rw |
| facet.locale.ca | facet.locale.hi | facet.locale.sa |
| facet.locale.ca_ES | facet.locale.hi_IN | facet.locale.sa_IN |
| facet.locale.cs | facet.locale.hr | facet.locale.sk |
| facet.locale.cs_CZ | facet.locale.hr_HR | facet.locale.sk_SK |
| facet.locale.da | facet.locale.hu | facet.locale.sl |
| facet.locale.da_DK | facet.locale.hu_HU | facet.locale.sl_SI |
| facet.locale.de | facet.locale.hy | facet.locale.sq |
| facet.locale.de_AT | facet.locale.hy_AM | facet.locale.sq_AL |
| facet.locale.de_BE | facet.locale.id | facet.locale.sr |
| facet.locale.de_CH | facet.locale.id_ID | facet.locale.sr_ME |
| facet.locale.de_DE | facet.locale.is | facet.locale.sr_RS |
| facet.locale.de_LI | facet.locale.is_IS | facet.locale.sv |
| facet.locale.de_LU | facet.locale.it | facet.locale.sv_SE |
| facet.locale.el | facet.locale.it_CH | facet.locale.ta |
| facet.locale.el_CY | facet.locale.it_IT | facet.locale.ta_IN |
| facet.locale.el_GR | facet.locale.ja | facet.locale.te |
| facet.locale.en | facet.locale.ja_JP | facet.locale.te_IN |
| facet.locale.en_AU | facet.locale.ka | facet.locale.th |
| facet.locale.en_BW | facet.locale.ka_GE | facet.locale.th_TH |
| facet.locale.en_CA | facet.locale.kk | facet.locale.tr |
| facet.locale.en_GB | facet.locale.kk_KZ | facet.locale.tr_TR |
| facet.locale.en_HK | facet.locale.kn | facet.locale.uk |
| facet.locale.en_IE | facet.locale.kn_IN | facet.locale.uk_UA |
| facet.locale.en_IN | facet.locale.ko | facet.locale.vi |
| facet.locale.en_MT | facet.locale.ko_KR | facet.locale.vi_VN |
| facet.locale.en_NZ | facet.locale.ks | facet.locale.zh |
| facet.locale.en_PH | facet.locale.ks_IN | facet.locale.zh_CN |
| facet.locale.en_SG | facet.locale.ku | facet.locale.zh_HK |
| facet.locale.en_US | facet.locale.ku_TR | facet.locale.zh_HK |
| facet.locale.en_ZW | facet.locale.ky | facet.locale.zh_SG |
| facet.locale.eo | facet.locale.ky_KG | facet.locale.zh_TW |
| facet.locale.es_AR | facet.locale.lg | |

# Chapter 8

## Modifying Package Manifests Programmatically

This chapter explains how package manifests can be machine edited to permit the automated annotation and checking of package manifests.

Chapter 4 covers the basics for how to publish packages. These techniques are all that is necessary to publish a package, but when publishing a large package, a large number of packages, or publishing packages over a period of time, there can be aspects which involve a significant time commitment for doing repetitive tasks.

For example, one rule that's used when publishing OpenIndiana is that all kernel modules should be tagged as requiring a reboot.

One option would be to impose this constraint through human examination and intervention, but that would be costly and likely error prone.

A second option would be to write a script or program that would handle tagging these actions. The difficulty here is that to be sure the program tagged actions correctly, it would need to parse the actions. This can certainly be done, but it duplicates a lot of functionality already in the IPS framework.

The third, and best, option is to use `pkgmogrify(1)`, provided by IPS, to transform the package manifests in repeatable ways.

There are two types of rules `pkgmogrify(1)` understands, transform and include. Transform rules are used to modify actions. Include rules cause other files to be processed.

### Transform Rules

When publishing OpenIndiana, we made the assumption that all files delivering in a subdirectory named "`kernel`" should be treated as kernel modules. This is the rule used to do the tagging:

```
<transform file path=.*kernel/.+ -> default reboot-needed true>
```

- The rule is enclosed with '`<`' and '`>`'. The portion of the rule to the left of the '`->`' is the selection section or matching section. The portion to the right of the '`->`' is the execution section of the operation.
- '`transform`' is the type of the rule
- '`file`' means this rule is for `file` actions
- '`path=.*kernel/.+`' means that only file actions with a path attribute that matches the regular expression '`.*kernel/.+`' are transformed
- '`default`' means that what follows should be added to a matching action unless a value for the attribute has already been set
- '`reboot-needed`' is the attribute being set
- '`true`' is the value of the attribute

The selection or matching section of a transform rule can restrict by action type and by action attribute values. The `pkgmogrify(1)` man page goes into detail about how these matching rules work, but the typical uses are for selecting actions which deliver to certain areas of the file system.

For example, a rule that began like this

```
<transform file dir link hardlink path=usr/bin.* -> [operation]>
```

could be used to ensure that `usr/bin` and everything delivered inside of it defaulted to the correct user or group. There is a long list of operations which `pkgmogrify(1)` can perform, detailed in the man page, which enable a package developer to programmatically add, remove, set, and edit actions' attributes as well as add and remove entire actions.

## Include Rules

Include rules enable transforms to be spread across multiple files and subsets reused by different manifests. Suppose a developer needs to deliver two packages: A and B. Both packages should have their `source-url` set to the same url, but only package B should have its files in `/etc` set to be `group=sys`.

In the manifest for package A, an include rule which pulls in the file with the `source-url` transform should be added. In package B, an include rule which pulls in the file containing the file `group` setting transform should be added.

Finally, an include rule which pulls in the file with the `source-url` transform should be added to either package B or the file with the transform that sets the group.

## Transform Order

Transforms are applied in the order in which they are encountered in a file. The ordering can be used to simplify the matching portions of transforms.

Suppose all files delivered in `/foo` should have a default group of `sys`, except those delivered in `/foo/bar`, which should have a default group of `bin`.

It's certainly possible to write a complex regular expression which matches all paths that begin with `/foo`, except those that begin with `/foo/bar`. Using the ordering of transforms makes it much simpler.

When ordering default transforms, always go from most specific to most general. Otherwise the latter rules will never be used.

In this case, the two rules would look like this:

```
<transform file path=foo/bar/.* -> default group bin>
<transform file path=foo/.* -> default group sys>
```

Using transforms to add an action using the matching described above would be difficult since the package developer would need to find a pattern which matched each package delivered once and only once.

`pkgmogrify(1)` creates synthetic actions to help with this issue. As `pkgmogrify(1)` processes manifests, for each manifest that sets the `pkg.fmri` attribute, a synthetic *pkg action* is created by `pkgmogrify(1)`. The package developer can match against the `pkg` action as if it was actually in the manifest.

For example, suppose a package developer wanted to add to every package an action containing the website (foo.com) where the source code for the delivered software can be found. The following transform accomplishes that:

```
<transform pkg -> emit set info.source-url=http://foo.com>
```

## Packaged Transforms

As a convenience to developers, a set of the transforms that were used when packaging OpenIndiana itself are available in `/usr/share/pkg/transforms`. At the time of writing, these are:

**developer**

Sets `facet.devel` on `*.h` header files delivered to `/usr/.*/include`, archive and lint libraries, `pkg-config(1)` data files, and `autoconf(1)` macros.

**documentation**

Sets a variety of `facet.doc.*` facets on documentation files.

**locale**

Sets a variety of `facet.locale.*` facets on files which are locale-specific.

**smf-manifests**

Adds a `restart_fmri` actuator pointing to the `svc:/system/manifest-import:default` on any packaged SMF manifests so that the system will import that manifest after the package is installed.

# Chapter 9

## Causing System Change With SMF

This chapter explains how to use the Service Management Facility (SMF) to automatically handle any necessary system changes that should occur as a result of package installation.

The package developer must determine which actions, when initially installed, updated or removed should cause a change to the system. For each of those actions, the package developer needs to determine which existing service provides the desired system change, or write a new service which provides the needed functionality and ensure that service is delivered to the system.

When the set of actions has been determined, those actions must be tagged in the package manifest with the correct *actuator* in order to cause that system change to occur.

As discussed in Chapter 1, some system changes are needed to employ the *software self-assembly* concept used by OpenIndiana and IPS, but system changes are not limited to this role.

We'll discuss the available actuators in the next section and then provide some examples.

### Actuators

The following tags can be added to any action in a manifest:

#### reboot-needed

This actuator takes the value `true` or `false`. This actuator declares that installation, removal or update of the tagged action requires a reboot when IPS is operating on a live image.

The following actuators are related to SMF services, and are the ones we will focus on in this chapter.

#### SMF Actuators

SMF actuators take a single SMF FMRI as a value, possibly including globbing characters to match multiple FMRIs. If the same FMRI is tagged by multiple actions, possibly across multiple packages being operated on, IPS will only trigger that actuator once.

The following list of SMF actuators describes the effect on the service FMRI that is the value of each named actuator:

**disable_fmri**
: The given service should be disabled prior to the package operation being performed

**refresh_fmri**
: The given service should be refreshed after the package operation has completed

**restart_fmri**
: The given service should be restarted after the package operation has completed

**suspend_fmri**
: The given service should be temporarily suspended prior to the package operation and enabled once it has completed

### Delivering an SMF Service

A package that delivers a new SMF service usually needs a system change. The package delivers the SMF manifest file and method script, and the packaged application requires that the SMF service it delivers must be available after package installation.

SVR4 post-install scripting could run an SMF command to restart the `svc:/system/manifest-import:default` service.

In IPS, the action delivering the manifest file into `lib/svc/manifest` or `var/svc/manifest` should instead be tagged with the actuator: `restart_fmri=svc:/system/manifest-import:default`.

The actuator ensures that when the manifest is added, updated, or removed, the `manifest-import` service is restarted, causing the service delivered by that SMF manifest to be added, updated, or removed.

If the package is added to a live-system, this action is performed once all packages have been added to the system during that packaging operation. If the package is added to an alternate boot environment, this action is performed during the first boot of that boot environment.

## A Service That Runs Once

Another common example is a system change that performs one-time configuration of the new software environment.

In the package delivering our application, we would include the following actions:

```
file path=opt/myapplication/bin/run-once.sh owner=root group=sys mode=0755
file path=lib/svc/manifest/application/myapplication-run-once.xml owner=root group=sys \
    mode=0644 restart_fmri=svc:/system/manifest-import:default
```

The SMF method script for the service could contain anything that is needed to further configure our application, or modify the system so that our application runs efficiently. In this example, we'll just have it write a simple log message.

Generally, we also want to ensure that the SMF service only performs work if the application has not already been configured. Another approach would be to package the service separate from the application itself, then have the method script remove the package that contains the service.

Our method script is:

```
#!/usr/bin/sh
. /lib/svc/share/smf_include.sh
assembled=$(/usr/bin/svcprop -p config/assembled $SMF_FMRI)
if [ "$assembled" == "true" ] ; then
    exit $SMF_EXIT_OK
fi
svccfg -s $SMF_FMRI setprop config/assembled = true
svccfg -s $SMF_FMRI refresh
echo "This is output from our run-once method script"
```

When testing a method script, it is advisable to run `pkg verify` before and after installing the package that runs the actuator. Compare the output of each run to ensure that the script doesn't attempt to modify any files that are not marked as editable.

Our SMF service manifest is:

```
<?xml version="1.0"?>
<!DOCTYPE service_bundle SYSTEM "/usr/share/lib/xml/dtd/service_bundle.dtd.1">

<service_bundle type='manifest' name='MyApplication:run-once'>

<service
    name='application/myapplication/run-once'
    type='service'
    version='1'>

    <single_instance />

    <dependency
        name='fs-local'
        grouping='require_all'
        restart_on='none'
        type='service'>
            <service_fmri value='svc:/system/filesystem/local:default' />
    </dependency>

    <dependent
            name='myapplication_self-assembly-complete'
            grouping='optional_all'
            restart_on='none'>
            <service_fmri value='svc:/milestone/self-assembly-complete' />
    </dependent>
    <instance enabled='true' name='default'>
            <exec_method
                type='method'
                name='start'
                exec='/opt/myapplication/bin/run-once.sh'
                timeout_seconds='0'/>

            <exec_method
                type='method'
                name='stop'
                exec=':true'
                timeout_seconds='0'/>

            <property_group name='startd' type='framework'>
                <propval name='duration' type='astring' value='transient' />
            </property_group>

            <property_group name='config' type='application'>
                <propval name='assembled' type='boolean' value='false' />
            </property_group>
    </instance>
</service>
</service_bundle>
```

Note that the SMF service has a `startd/duration` property set to `transient` so that `svc.startd(1M)` doesn't track processes for this service. Also note that it adds itself as a dependency to the `self-assembly-complete` system milestone.

## Self-Assembly Hints

Here are some additional hints when writing SMF methods to support self-assembly:

### Timestamps

In an SMF method script, it can be efficient to use the output of `ls -t` on a directory of packaged configuration file fragments, using `head -1` to select the most recently changed version. The timestamp of this file can be compared with the timestamp of the unpackaged configuration file which is compiled from those fragments. This comparison can be used when deciding whether the service needs to recompile the configuration file.

This can be useful if the process of compiling a configuration file from those fragments is expensive to perform each time the method script runs.

### Timeouts

In the example SMF service used in this chapter, we had a `timeout_seconds` value of `0` for the start method. This means that SMF will wait indefinitely for self-assembly to complete.

Depending on circumstances, developers might want to impose a finite timeout on their self-assembly processes, enabling SMF to drop the service to `maintenance` if something goes wrong. This can assist the developer when debugging.

# Chapter 10

# Advanced Update

This chapter deals with more complex package update issues, and describes several features in IPS designed to simplify these problems.

For most update operations, IPS will automatically do exactly what is needed to install updated packages. There are some cases, however, that require the developer to provide additional information to IPS.

For performance reasons, the solver works purely on the dependency information included in packages. Packages whose dependencies indicate that they can be installed at the same time but whose content conflicts cause conflict checking to fail in pre-installation.

An example of conflicting content is two packages installing the same file. If conflict checking fails, the user must try different package versions and then manually specify acceptable versions.

Ensuring that conflicting packages cannot be installed due to constraining dependencies is a responsibility of the package developer. As mentioned in Chapter 4, `pkglint(1)` can assist with this task.

## Renaming, Merging and Splitting Packages

Often, the desired organization of a software component changes, whether because of mistakes in the original packages, changes in the product or its usage over time, or changes in the surrounding software environment. Also, sometimes just the name of a package needs to change. When contemplating such changes, thought must be given to the customer who is upgrading their system to ensure that unintended side effects do not occur.

Three types of package reorganization are discussed in this section, in order of increasingly complex considerations for pkg update:

1. Renaming single packages
2. Merging two packages
3. Splitting a package

### Renaming a Single Package

Simple renames are straightforward. IPS provides a mechanism to indicate that a package has been renamed. To rename a package, publish a new version of the existing package with the following two actions:

- A `set` action in the following form:

```
set name=pkg.renamed value=true
```

- A `require` dependency on the new package

A renamed package cannot deliver contents other than depend or set actions.

The new package **must** ensure that it cannot be installed at the same time as the original package before the rename. If both packages are covered by the same incorporation dependency, this is automatic.

If not, the new package must contain an `optional` dependency on the old package at the renamed version. This ensures that the solver will not select both packages, which would fail conflict checking.

Anyone installing this renamed package will automatically receive the new named package, since it is a dependency of the old version. If a renamed package is not depended upon by any other packages, it is automatically removed from the system. The presence of older software can cause a number of renamed packages to be shown as `installed`; when that older software is removed the renamed packages are automatically removed as well.

Packages can be renamed multiple times without issue, although this is not recommended as it can be confusing to users.

### Merging Two Packages

Merging packages is straightforward as well. The following two cases are examples of merging packages:

- One package absorbs another package at the renamed version.
- Two packages are renamed to the same new package name.

### One Package Absorbs Another

Suppose package *A@2* will absorb package *B@3*. Simply rename package *B* to package *A@2*; remember to include an optional dependency in *A@2* on *B@3* unless both packages are incorporated so they update in lockstep as above. A user upgrading *B* to *B@3* will now get *A* installed, which has absorbed *B*.

### Two Packages Are Renamed

In this case, simply rename both packages to the name of the new merged package, including two `optional` dependencies on the old packages in the new one if they are not otherwise constrained.

### Splitting a Package

When you split a package, rename each resulting new package as described in Renaming a Single Package. If one of the resulting new packages is not renamed, the pre-split and post-split versions of that package are not compatible and might violate dependency logic when the end user tries to update the package.

Rename the original package, including multiple `require` dependencies on all new packages that resulted from the split. This ensures that any package that had a dependency on the original package will get all the new pieces.

Some components of the split package can be absorbed into existing packages as a merge. See One Package Absorbs Another.

## Obsoleting Packages

Package obsoletion is the mechanism by which packages are emptied of contents and are removed from the system. Such a package does not satisfy `require` dependencies, so an installed package with a `require` dependency on a package that becomes obsolete will prevent update unless a newer version of the installed package is available that does not contain the `require` dependency.

A package is made obsolete by publishing a new version with no content except for the following `set` action:

```
set name=pkg.obsolete value=true
```

A package can be made non-obsolete by publishing newer versions. Users who updated through the obsoletion will lose this package, while those who did not will not.

## Preserving Editable Files During Package Renaming or Path Changes

One common issue with updating packages is the migration of editable files, either in the file system or between packages. IPS attempts to migrate editable files that move between packages (for example, as the result of a rename) if the file is not renamed and the path of the file has not changed. However, if the path changes, the following must be done for the user's customizations to be preserved:

If the `file` action in the old package does not contain the attribute `original_name`, that attribute must be added. Set the value to the original name of the package, followed by a colon and then the path to the file without a leading

'/'. Once this is present on an editable file, it must not be changed. This value acts as a unique identifier for all moves going forward so that regardless of the number of versions skipped on an update, the user's content is properly preserved.

## Moving Unpackaged Contents on Directory Removal or Rename

Normally, unpackaged contents are salvaged when the containing directory is removed, because the last reference to it disappears.

When a directory changes names, the packaging system treats this as the removal of the old directory and the creation of a new one. Any editable files that are still in the directory when the directory is renamed or removed are salvaged.

If the old directory has unpackaged content such as log files that should be moved to the new directory, this can be done with the `salvage-from` attribute if placed on the new directory.

For example, suppose we want to rename a directory from:

```
/opt/mydata/log
```

to:

```
/opt/yourdata/log
```

In the same package version that removes the former directory and introduces the latter directory, include the following attribute on the `dir` action that creates `/opt/yourdata/log`:

```
salvage-from=opt/mydata/log
```

Any unpackaged contents of any time are migrated to the new location.

The `salvage-from` attribute is covered later in this chapter, when discussing data that should be shared between boot environments.

## Delivering Multiple Implementations of a Given Application

In some cases, it can be desirable to deliver multiple implementations of a given application, having all implementations available on the system, but with one implementation set as the *preferred* implementation.

The preferred implementation would have symlinks to its binaries installed, say, to `/usr/bin` for ease of discovery. We would also like to allow the administrator to change the preferred implementation as required, without having to add or remove any additional packages.

A good example of this would be where we have several versions of GCC installed, each in their own package, but would like `/usr/bin/gcc` to always point to our preferred version.

IPS uses the concept of *mediated links* for this purpose. A mediated link is a symbolic link that is controlled by the `pkg set-mediator` and `pkg unset-mediator` commands, documented in the `pkg(1)` man page.

The `link` actions in the packages that deliver different implementations of that application are said to participate in a *mediation*.

The following attributes can be set on `link` actions to control how mediated links are delivered:

**mediator**

Specifies the entry in the mediation namespace shared by all path names participating in a given mediation group (for example `python`).

Link mediation can be performed based on `mediator-version` and `mediator-implementation`. All mediated links for a given path name must specify the same `mediator`. However, not all mediator versions and implementations need to provide a link at a given path. If a mediation does not provide a link, then the link is removed when that mediation is selected.

A mediator, in combination with a specific version and/or implementation represents a *mediation* that can be selected for use by the packaging system.

**mediator-version**

Specifies the version (expressed as a dot-separated sequence of non-negative integers) of the interface described by the `mediator` attribute. This attribute is required if `mediator` is specified and `mediator-implementation` is not. A local system administrator can explicitly set the version to use. The value specified should generally match the version of the package delivering the link (for example, `runtime/python-26` should use `mediator-version=2.6`), although this is not required.

**mediator-implementation**

Specifies the implementation of the mediator for use in addition to or instead of the `mediator-version`. Implementation strings are not considered to be ordered. A string is arbitrarily selected by `pkg(5)` if not explicitly specified by a system administrator.

The value can be a string of arbitrary length composed of alpha-numeric characters and spaces. If the implementation itself can be or is versioned, then the version should be specified at the end of the string, after a '@' (expressed as a dot-separated sequence of non-negative integers). If multiple versions of an implementation exist, the default behavior is to select the implementation with the highest version.

If only one instance of an implementation-mediation link at a particular path is installed on a system, then that one is chosen automatically. If future links at the path are installed, the link will not be switched unless a vendor, site, or local override applies, or if one of the links is version-mediated.

**mediator-priority**

When resolving conflicts in mediated links, `pkg(5)` normally chooses the link with the greatest value of `mediator-version` or based on `mediator-implementation` if that is not possible. This attribute is used to specify an override for the normal conflict resolution process.

If this attribute is not specified, the default mediator selection logic is applied.

- If the value is `vendor`, the link is preferred over those that do not have a `mediator-priority` specified.
- If the value is `site`, the link is preferred over those that have a value of `vendor` or that do not have a `mediator-priority` specified.

A local system administrator can override the selection logic described above.

Here are two sample manifests that participate in a mediation for the link `/usr/bin/myapp`:

```
set name=pkg.fmri value=pkg://test/myapp-impl-1@1.0,5.11:20111021T035233Z
file path=usr/myapp/5.8.4/bin/myapp group=sys mode=0755 owner=root
link path=usr/bin/myapp target=usr/myapp/5.8.4/bin/myapp mediator=myapp mediator-version=5.8.4
```

```
set name=pkg.fmri value=pkg://test/myapp-impl-2@1.0,5.11:20111021T035239Z
file path=usr/myapp/5.12/bin/myapp group=sys mode=0755 owner=root
link path=usr/bin/myapp target=usr/myapp/5.12/bin/myapp mediator=myapp mediator-version=5.12
```

We can install both of these packages to the same image:

```
$ pkg list myapp-impl-1 myapp-impl-2
NAME (PUBLISHER)                                  VERSION                    IFO
myapp-impl-1                                      1.0                        i--
myapp-impl-2                                      1.0                        i--
```

Using the `pkg mediator` command, we can see the mediations in use:

```
$ pkg mediator
MEDIATOR VER. SRC. VERSION IMPL. SRC. IMPLEMENTATION
myapp    local     5.12    system
$ ls -al usr/bin/myapp
lrwxrwxrwx   1 root     sys         23 Oct 21 16:58 usr/bin/myapp -> usr/myapp/5.12/bin/myapp
```

We can see which other packages participate in the `myapp` mediation using `pkg search`:

```
$ pkg search -ro path,target,mediator,mediator-version,pkg.shortfmri ::mediator:myapp
PATH          TARGET                     MEDIATOR MEDIATOR-VERSION PKG.SHORTFMRI
usr/bin/myapp usr/myapp/5.12/bin/myapp  myapp    5.12             pkg:/myapp-impl-2@1.0
usr/bin/myapp usr/myapp/5.8.4/bin/myapp myapp    5.8.4            pkg:/myapp-impl-1@1.0
```

We can also change the mediation as desired:

```
# pkg set-mediator -V 5.8.4 myapp
            Packages to update:  2
           Mediators to change:  1
       Create boot environment: No
Create backup boot environment: No


PHASE                                        ITEMS
Indexing Packages                              2/2
PHASE                                      ACTIONS
Update Phase                                   1/1

PHASE                                        ITEMS
Image State Update Phase                       2/2
Reading Existing Index                         8/8
Indexing Packages                              2/2

# ls -al usr/bin/myapp
lrwxrwxrwx   1 root     sys         24 Oct 21 17:02 usr/bin/myapp -> usr/myapp/5.8.4/bin/myapp
```

## Delivering Directories To Be Shared Across Boot Environments

In general, IPS doesn't support delivery of packaged contents to datasets that span boot environments (BEs). This is because such shared contents, if updated in one boot environment, might not meet the definitions for other boot environments. For example, we could foresee a case where a `pkg verify` of packaged content that was delivered with different attributes by packages in two separate boot environments, yet shared between them, would result in in errors.

However, some of the unpackaged files (the files stored in the file system that were not delivered by any IPS package) found in a boot environment must be shared across boot environments to preserve normal system operation in the face of multiple boot environments.

Some examples include `/var/mail`, `/var/log` and the like. Customers are likely to place such data on separate datasets as well, or on remote file servers. However, creating per-directory datasets would mean that many datasets would be created per zone, which is not desirable.

The goal can be achieved using a shared dataset, mounted into the BE during boot, with symbolic links from locations inside the BE pointing into that dataset. Inside the BE, applications deliver primordial directory structure to a *.migrate* staging directory.

As noted above, no packaged file content should be shared between boot environments, furthermore, it is not possible or desirable to share any file system objects other than files.

Update is supported from older versions of a package that did not share content. Use a `salvage-from` attribute as discussed in Moving Unpackaged Contents on Directory Removal or Rename and shown in the example below.

The package should no longer deliver the old directory.

During boot, a script can be run as part of an SMF method script to move file content from the *.migrate* directory into the shared dataset. This script is responsible for recreating the directory structure that it finds under the *.migrate* directory in the boot environment, and moving file contents from the *.migrate* directory to the shared dataset.

For example, for a package that previously delivered the action:

```
dir path=opt/myapplication/logs owner=daemon group=daemon mode=0755
```

we first create a dataset `rpool/OPTSHARE` (which can be used by other shared content from `/opt`) This dataset creation could alternatively be done by the SMF method script during boot:

```
# zfs create rpool/OPTSHARE
# zfs set mountpoint=/opt/share rpool/OPTSHARE
```

A package can then deliver a symbolic link from their previously packaged directory to an as-yet nonexistent target beneath `/opt/share`:

```
link path=opt/myapplication/logs target=../../opt/share/myapplication/logs
```

Packages can now deliver the directory into this *.migrate* area:

```
dir path=opt/.migrate/myapplication/logs owner=daemon group=daemon \
    mode=0755 reboot-needed=true salvage-from=/opt/myapplication/logs
```

We use the `salvage-from` attribute to move files from the old location into the *.migrate* directory.

We require a `reboot-needed` actuator for these directory entries in order to properly support updates of Immutable Zones mentioned in Chapter 1, which boot as far as the `svc:/milestone/self-assembly-complete:default` milestone in read/write mode if self-assembly is required, before rebooting read-only. See the discussion of `file-mac-profile` in the `zonecfg(1M)` manual page for more on Immutable Zones.

Our SMF service, on reboot, will then move any salvaged directory content into the shared dataset, and the symbolic links from `/opt/myapplication` point into that shared dataset.

# Chapter 11

## Signing Packages

One important consideration in the design of IPS was being able to validate that the software installed on the customer's machine was actually as originally specified by the publisher. This ability to validate the installed system is key for both the customer and the support engineering staff.

To support this validation, manifests can be signed in IPS with the signatures becoming part of the manifest. Signatures are represented as actions like all other manifest content. Since manifests contain all the package metadata - file permissions, ownership, content hashes, etc., a `signature` action that validates that the manifest has not be altered since it was published is an important part of system validation.

The `signature` actions form a tree that includes the delivered binaries such that complete verification of the installed software is possible.

There are other uses for manifest signing beyond validation; signatures can also be used to indicate approval by other organizations or parties.

For example, the internal QA organization could sign manifests of packages once it was determined the packages were qualified for production use. Policy could mandate such approvals prior to installation.

As a result, a useful characteristic for signatures is to be independent of other signatures in a manifest. Signatures can be added or removed without invalidating the other signatures that might be present. This feature also facilitates production hand offs, with signatures used along the path to indicate completion along the way. Subsequent steps can optionally remove previous signatures at any time without ill effect.

`signature` actions look like this:

```
signature <hash of certificate> algorithm=<signature algorithm> \
    value=<signature value> \
    chain="<hashes of certs needed to validate primary certificate>" \
    version=<pkg version of signature>
```

The payload and `chain` attributes represent the packaging hash of the PEM (Privacy Enhanced Mail) files, containing the x.509 certificates downloadable from the originating repository. The value is the signed hash of the manifest's message text, prepared as discussed below. The payload certificate is the certificate which verifies the value in `value`.

The other certificates presented needs to form a certificate path that leads from the payload certificate to the trust anchors that were established as part of the publisher configuration.

Two types of signature algorithms are currently supported. The first is the RSA group of signature algorithms; an example is `rsa-sha256`. The bit after the dash specifies the hash algorithm to use to change the message text into a single value the RSA algorithm can use.

The second type of signature algorithm is compute the hash only. This type of algorithm exists primarily for testing and process verification purposes and presents the hash as the signature value. A signature action of this type is indicated by the lack of a payload certificate hash. This type of signature action is verified if the image is configured to check signatures. Its presence however does not count as a signature if signatures are required:

```
signature algorithm=<hash algorithm> value=<hash> \
    version=<pkg version of signature>
```

Additional metadata can be added to a signature if desired, as with any other action. Such metadata is protected by that signature.

Policies can be set for the image or for specific publishers. The policies include ignoring signatures, verifying existing signatures, requiring signatures, and requiring that specific common names must be seen in the chain of trust. Other policies might be added in the future.

Publishing a signed manifest is a two step process:

1. Publish the package unsigned to a repository.
2. Update the package in place, using `pkgsign(1)` to append a signature action to the manifest in the repository.

This process leaves the package intact, including its timestamp.

This process enables a signature action to be added by someone other than the publisher without invalidating the original publisher's signature. For example, the QA department of a company might want to sign all packages that are installed internally to indicate they have been approved for use, but not republish the packages, which would create a new timestamp and invalidate the signature of the original publisher.

Note that `pkgsign(1)` is the only way to publish a signed package. If one attempts to publish a package already containing a signature, that signature is removed and a warning is emitted. The `pkgsign(1)` man page contains examples of how to use `pkgsign(1)`.

One current restriction to be aware of is that signature actions with variants are ignored. That means that doing a `pkgmerge(1)` on a pair of manifests will invalidate any signatures which were previously applied. Signing the package should be the last step of the package development before the package is tested.

`pkgsign(1)` does not perform all the possible checks for its inputs when signing packages. This means it's important to check signed packages to ensure that they can be properly installed after being signed. What follows are some of the errors that can appear when attempting to install or update a signed package along with explanations of what the errors mean and how to solve the problem.

## Errors Involving Signed Packages

A signed package can fail to install or update for reasons that are unique to signed packages. For example, if a package's signature fails to verify, or if its chain of trust can't be verified or anchored to a trusted certificate, the package will fail to install.

When installing signed packages, certain image properties will influence the checks that are performed on packages. These properties are:

- `signature-policy`
- `signature-required-names`
- `trust-anchor-directory`

See the `pkg(1)` man page for further information about these properties, and their permitted values.

What follows are some examples of different failure paths and what can be done to resolve them.

#### Example 1: Chain Certificate Not Found

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta3/emailAddress=cs1_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3

The package involved is:pkg://test/example_pkg@1.0,5.11-0:20110919T184152Z
```

The error shown above happens when a certificate in the chain of trust is missing or otherwise erroneous.

In this case, there were three certificates in the chain of trust when the package was signed. It was rooted in the trust anchor, a certificate named `ta3`. `ta3` signed a chain cert named `ch1_ta3`.

`ch1_ta3` signed a code signing certificate named `cs1_ch1_ta3`. When `pkg(1)` tried to install the package, it was able to locate the code signing certificate, `cs1_ch1_ta3`, but it couldn't locate the chain certificate, `ch1_ta3`, so the chain of trust could not be established.

The most common cause of this problem is failing to provide the right certificate(s) to the `-i` option of `pkgsign(1)`.

#### Example 2: Authorized Certificate Not Found

```
pkg install: The certificate which issued this certificate:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_cs8_ch1_ta3/emailAddress=cs1_cs8_ch1_ta3
could not be found. The issuer is:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs8_ch1_ta3/emailAddress=cs8_ch1_ta3
The package involved is:pkg://test/example_pkg@1.0,5.11-0:20110919T201101Z
```

The error shown above is similar to the error in *Example 1* but has a different cause.

In this case, the package was signed using the `cs1_cs8_ch1_ta3` certificate, which was signed by the `cs8_ch1_ta3` certificate.

The problem is that the `cs8_ch1_ta3` certificate wasn't authorized to sign other certificates. (To be specific, the `cs8_ch1_ta3` certificate had the `basicConstraints` extension set to `CA:false` and marked critical.)

When `pkg(1)` verifies the chain of trust, it doesn't find a certificate which was allowed to sign the `cs1_cs8_ch1_ta3` certificate. Since the chain of trust can't be verified from the leaf to the root, `pkg(1)` prevents the package from being installed.

#### Example 3: Untrusted Self-Signed Certificate

```
pkg install: Chain was rooted in an untrusted self-signed certificate.
The package involved is:pkg://test/example_pkg@1.0,5.11-0:20110919T185335Z
```

The error shown above happens when a chain of trust ends in a self-signed certificate which isn't trusted by the system.

When a developer creates a chain of certificates using `openssl` for testing, the root certificate is usually self-signed, since there's little reason to have an outside company verify a certificate only used for testing.

In a test situation, there are two solutions.

The first is to add the self-signed certificate which is the root of the chain of trust into `/etc/certs/CA` and refresh the `system/ca-certificates` service.

This mirrors the likely situation customers will encounter where a production package is signed with a certificate that's ultimately rooted in a certificate that's delivered with the operating system as a trust anchor.

The second solution is to approve the self-signed certificate for the publisher which offers the package for testing by using the `--approve-ca-cert` option for the `set-publisher` subcommand to `pkg(1)`.

### Example 4: Signature Value Does Not Match Expected Value

```
pkg install: A signature in pkg://test/example_pkg@1.0,5.11-0:20110919T195801Z
could not be verified for this reason:
The signature value did not match the expected value. Res: 0
The signature's hash is 0ce15c572961b7a0413b8390c90b7cac18ee9010
```

The error shown above happens when the value on the signature action could not be verified using the certificate which the action claims was paired with the key used to sign the package.

There are two possible causes for an error like this.

The first is that the package has been changed since it was signed. This is unlikely to happen since `pkgsend(1)` will strip existing signature actions during publication (since the new timestamp the package will get will invalidate the old signature) but is possible if the package's manifest has been hand edited since signing.

The second, and most likely cause, is that the key and certificate used to the sign the package weren't a matched pair. If the certificate given to the `-c` option of `pkgsign(1)` wasn't created with the key given to the `-k` option of `pkgsign(1)`, the package is signed, but its signature won't be verified.

### Example 5: Unknown Critical Extension

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs2_ch1_ta3/emailAddress=cs2_ch1_ta3
could not be verified because it uses a critical extension that pkg5 cannot
handle yet. Extension name:issuerAltName
Extension value:<EMPTY>
```

The error above happens when a certificate in the chain of trust uses a critical extension which `pkg(1)` doesn't understand.

Until `pkg(1)` learns how to process that critical extension, the only solution is to regenerate the certificate without the problematic critical extension.

### Example 6: Unknown Extension Value

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs5_ch1_ta3/emailAddress=cs5_ch1_ta3
could not be verified because it has an extension with a value that pkg(5)
does not understand.
Extension name:keyUsage
Extension value:Encipher Only
```

The error above is similar to the error in *Example 5* except that the problem is not with an unfamiliar critical extension but with a value that `pkg(1)` doesn't understand for an extension which `pkg(1)` does understand.

In this case, `pkg(1)` understands the `keyUsage` extension, but doesn't understand the value '`Encipher Only`.' The error will look the same whether the extension in question is critical or not.

The solution, until `pkg(1)` learns about the value in question, is to remove the value from the extension, or remove the extension entirely.

### Example 7: Unauthorized Use of Certificate

```
pkg install: The certificate whose subject is
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=ch1_ta3/emailAddress=ch1_ta3
could not be verified because it has been used inappropriately.
The way it is used means that the value for extension keyUsage must include
'DIGITAL SIGNATURE' but the value was 'Certificate Sign, CRL Sign'.
```

The error above occurs when a certificate has been used for a purpose for which it was not authorized.

In this case, the certificate `ch1_ta3` has been used to sign the package. It's `keyUsage` extension means that it's only valid to use that certificate to sign other certificates and CRL's.

### Example 8: Unexpected Hash Value

```
pkg install: Certificate
/tmp/ips.test.7149/0/image0/var/pkg/publisher/test/certs/0ce15c572961b7a0413b8390c90b7cac18ee9010
has been modified on disk. Its hash value is not what was expected.
```

The error above means what it says.

The certificate at the provided path is used to verify the package being installed but the hash of the contents on disk don't match what the signature action thought they should be.

This indicates that the certificate has been changed since it was last retrieved from the publisher.

The simple solution is to remove the certificate and allow `pkg(1)` to download the certificate again.

### Example 9: Revoked Certificate

```
pkg install: This certificate was revoked:
/C=US/ST=California/L=Menlo Park/O=pkg5/CN=cs1_ch1_ta4/emailAddress=cs1_ch1_ta4
for this reason: None
The package involved is: pkg://test/example_pkg@1.0,5.11-0:20110919T205539Z
```

The error above indicates the certificate in question, which was in the chain of trust for the package to be installed, was revoked by the issuer of the certificate.

# Chapter 12

## Handling Non-Global Zones

This chapter describes how IPS handles zones and discusses those cases where package developers should be aware of zones.

### Packaging Considerations For Non-Global Zones

Developing packages which work consistently with zones usually involves little to no additional work. However, a few situations call for close attention from developers. When considering zones and packaging there are two questions which need to be answered:

- Does anything in my package have an interface which crosses the boundary between the global zone and non-global zones?
- How much of the package should be installed in the non-global zone?

#### Does The Package Cross the Global, Non-Global Zone boundary?

If a package delivers both kernel and userland functionality, and both sides of that interface must be updated accordingly, then the package must be updated in any zones that contain that package whenever the package in the non-global zone gets updated.

This can be done using a `parent` dependency in the package being developed. If a single package delivers both sides of the interface, then a `parent` dependency on `feature/package/dependency/self` will ensure that the global zone and the non-global zones contain the same version of the package, preventing version skew across the interface.

The dependency will also ensure that if the package is in a non-global zone, then it is also present in the global zone.

If the interface spans multiple packages, then the package containing the non-global zone side of the interface must contain a `parent` dependency on the package which delivers the global zone side of the interface. The `parent` dependency is also discussed in Chapter 6.

#### How Much of a Package Should Be Installed in a Non-Global Zone?

If the answer to this question is "all of it" (and that's typically the case) then nothing needs to be done to the package to enable it to function properly.

For consumers of the package, though, it can be reassuring to see that the package author properly considered zone installation and decided that this package can function in a zone.

For that reason, developers should explicitly state that their package functions in both global and non-global zones. This is done by adding the following action to the manifest:

```
set name=variant.opensolaris.zone value=global value=nonglobal
```

If no content in the package can be installed in a non-global zone (for example a package which only delivers kernel modules or drivers), then the package should specify that it cannot be installed in a zone. This is done by adding the following action to the manifest:

```
set name=variant.opensolaris.zone value=global
```

If some but not all of the content in the package can be installed in a non-global zone, then take the following steps:

1. Use the following `set` action to state that the package can be installed in both global and non-global zones:

```
set name=variant.opensolaris.zone value=global value=nonglobal
```

2. Identify the actions that are only relevant in either the global or non-global zone. The global-zone-only actions should have the attribute `variant.opensolaris.zone=global`. Similarly, actions that only apply in non-global zones should have the attribute `variant.opensolaris.zone=nonglobal`.

If a package has a `parent` dependency or has pieces which are different in global and non-global zones, it's important to test that the package works as expected in the non-global zone as well as the global zone. If the package has a `parent` dependency on itself, then the global zone should configure the repository which delivers the package as one of its origins. The package should be installed in the global zone first, and then in the non-global zone for testing.

## Troubleshooting Zones

Occasionally problems might be encountered when trying to install the package in the non-global zone.

Typically the first steps to take to attack the problem are to ensure that the following services are online in the global zone:

- `svc:/application/pkg/zones-proxyd:default`
- `svc:/application/pkg/system-repository:default`

and that the following service is online in the non-global zone:

- `svc:/application/pkg/zones-proxy-client:default`

These three services provide publisher configuration to the non-global zone and a communication channel that the non-global zone can use to make requests to the repositories assigned to the system publishers served from the global zone.

Remember that you won't be able to update the package in the non-global zone, since it has a `parent` dependency on itself. Initiating the update from the global zone and allowing the linked image code in `pkg(1)` to update the non-global zone is the right solution.

Once the package is installed in the non-global zone, testing its functionality can begin.

If the package does not have a `parent` dependency on itself, then it's not necessary to configure the publisher in the global zone nor install the package there. Further, updating the package in the global zone will not update it in the non-global zone, causing potentially unexpected results when testing the older non-global zone package.

The simplest solution in this situation is to make the publisher available to the non-global zone and install and update the package from within the zone.

If the zone cannot access the publisher's repositories, then configuring the publisher in the global zone will allow the `zones-proxy-client` and `system-repository` services to proxy access to the publisher for the non-global zone. In that case, it's still best to install and update the package in the non-global zone.

# Chapter 13

## How IPS Features Are Used when Packaging the OpenIndiana OS

This chapter describes how IPS is used to package OpenIndiana, and how the various dependency types are used to define working package sets for the OS.

We include this chapter to give another concrete example of how IPS can be used to manage a complex set of software, and talk about some of the IPS features that were used.

### Versioning

In Chapter 3 we discussed the `pkg.fmri` attribute, and talked about the different components of the version field, describing how the version field can be used to support different models of software development.

This section explains how OpenIndiana uses the version field, and is provided to give an insight into the reasons why a fine-grained versioning scheme can be useful. Developers **do not** have to follow the same versioning scheme as OpenIndiana.

Given a sample package:

```
pkg://openindiana.org/system/library@0.5.11,5.11-2018.0.0.18233:20190417T022656Z
```

This is how the version field `0.5.11,5.11-2018.0.0.18233:20190417T022656Z` is broken down:

**0.5.11**

The component version. For packages that are provided by illumos-gate, this is the OS major.minor version. For packages developed outside, this is the upstream version. For example, the Apache Web Server in the package:

`pkg:/web/server/apache-24@2.4.39,5.11-2018.0.0.0:20190406T083404Z`

has the component version 2.4.39.

**5.11**

This is the build version. This is used to define the OS release that this package was built for and should always be 5.11 for packages created for OpenIndiana.

**2018.0.0.18233**

This is the branch version. OpenIndiana uses the following notation for the branch version in this release:

- 2018: Major release number. Usually corresponds to the current year
- 0: Release minor number. Can be incremented on significant updates.
- 0: Update number. Usually incremented when a lot of components need rebuilding.
- 18233: Component revision, incremented for each component update. In this case it refers to illumos-gate commit number.

**20190417T022656Z**

This is the timestamp, defined when the package was published.

### Incorporations

OpenIndiana is delivered by a set of packages, with each group of packages constrained by an incorporation.

Each incorporation roughly represents the organization that developed each group of packages, though there are some cross-incorporation dependencies within the packages themselves. The following is a list of the incorporation packages in OpenIndiana:

- `pkg:/consolidation/cde/cde-incorporation`
- `pkg:/consolidation/dbtg/dbtg-incorporation`
- `pkg:/consolidation/install/install-incorporation`
- `pkg:/consolidation/jdmk/jdmk-incorporation`
- `pkg:/consolidation/man/man-incorporation`
- `pkg:/consolidation/nspg/nspg-incorporation`
- `pkg:/consolidation/osnet/osnet-incorporation`
- `pkg:/consolidation/userland/userland-incorporation`
- `pkg:/consolidation/X/X-incorporation`

Each of these incorporations includes:

- general package metadata
- `incorporate` dependencies, sometimes with `variant.arch` variants to denote dependencies that are specific to a given architecture
- a `license` action that ensures that when the incorporation is installed, a license is displayed

Each of the packages delivered on the system contains a `require` dependency on one of these incorporations.

OpenIndiana also includes a special package called `entire`.

The `entire` constrains dependency on userland-incorporation.

## facet.version-lock.*

Some of the incorporations, listed above use `facet.version-lock.*` facets, which were discussed in Chapter 6.

For example, looking at the `pkg:/consolidation/userland/userland-incorporation` package, we see:

```
.
.
depend type=incorporate \
    fmri=library/python/subversion@1.9.7-2018.0.0.1 \
    depend facet.version-lock.library/python/subversion=true
depend type=incorporate \
    fmri=library/security/libassuan@2.1.3-2018.0.0.1 \
    facet.version-lock.library/security/libassuan=true
depend type=incorporate \
    fmri=network/chat/ircii@0.2011.11.15-2018.0.0.1 \
    facet.version-lock.network/chat/ircii=true
.
.
etc.
```

enabling the administrator to allow certain packages to float free from the constraints of the incorporation package.

Notably, the `entire` package also contains version-lock facet, allowing userland incorporation to be removed. However, this can result in a system which is difficult to fix, and this package should only be unlocked on development systems.

## Informational attributes

The following attributes are not necessary for correct package installation, but having a shared convention lowers confusion between publishers and users.

**info.classification**

See Chapter 3 under "Set actions", and Appendix A.

**info.keyword**

A list of additional terms that should cause this package to be returned by a search.

**info.maintainer**

A human readable string describing the entity providing the package. For an individual, this string is expected to be their name, or name and email.

**info.maintainer-url**

A URL associated with the entity providing the package.

**info.upstream**

A human readable string describing the entity that creates the software. For an individual, this string is expected to be their name, or name and email.

**info.upstream-url**

A URL associated with the entity that creates the software delivered within the package.

**info.source-url**

A URL to the source code bundle, if appropriate, for the package.

**info.repository-url**

A URL to the source code repository, if appropriate, for the package.

**info.repository-changeset**

A changeset ID for the version of the source code contained in info.repository-url.

## OpenIndiana Attributes

**org.opensolaris.arc-caseid**

One or more case identifiers (e.g., PSARC/2008/190) associated with the ARC case (Architecture Review Committee) or cases associated with the component delivered by the package.

**org.opensolaris.smf.fmri**

One or more FMRIs representing SMF services delivered by this package. These attributes are automatically generated by `pkgdepend(1)` for packages containing SMF service manifests.

## OpenIndiana Tags

**variant.opensolaris.zone**

See Chapter 12

## Organization Specific Attributes

Organizations wanting to provide a package with additional metadata or to amend an existing package's metadata (in a repository that they have control over) must use an organization-specific prefix. For example, a service organization might introduce `service.example.com,support-level` or `com.example.service,support-level` to describe a level of support for a package and its contents.

# Chapter 14

## Republishing Packages

This chapter describes how administrators can modify existing packages for local conditions.

Occasionally administrators need to override attributes or modify packages they did not produce. This might be to replace a portion of the package with an internal implementation, or something as simple as removing binaries not permitted on systems.

While other packaging systems provide various mechanisms to "force" installation, IPS focuses instead on making it easy to republish an existing package with the desired modifications. This makes upgrade much easier since new versions can be re-published with the same modifications. It also enables the rest of the packaging system to function normally since instead of forcing IPS to ignore changes, packages reflect the correct, installed state of the system.

Of course, running a system with a republished package can cause issues with the support organization if any connection is suspected between observed problems and the modified package.

The basic steps are as follows:

1. Use `pkgrecv(1)` to download the package to be re-published in a *raw* format to a specified directory. All of the files are named by their hash value, and the manifest is named `manifest`. Remember to set any required proxy configuration in the `http_proxy` environment variable.
2. Use `pkgmogrify(1)` to modify the manifest in the desired manner. Any timestamp from the internal package FMRI should be removed to prevent confusion during publication as it is ignored.

   If changes are significant, running the resulting package through `pkglint(1)`, as shown in Chapter 4, is a good idea.
3. Republish the package with `pkgsend(1)`. Note that this republication will strip the package of any signatures that are present and will ignore any timestamp specified by `pkg.fmri`. To prevent a warning message you might want to remove signature actions in the `pkgmogrify(1)` step.

   If the administrator doesn't have permission to publish to the original source of the package, they can create a repository with `pkgrepo(1)`, then use `pkg set-publisher --search-before=<original>` to have the client look for packages from the new repository before falling back to the original publisher.
4. Optionally, sign the package using `pkgsign(1)` so that internal processes can be followed. Packages should be signed before they are installed (even during testing) to prevent client caching issues.

## Example 1: Change Package Metadata

Here's a simple example, where we change the `pkg.summary` field to be ‘"IPS has lots of features"’ instead of whatever was there originally, and republish to our new repository:

```
$ mkdir republish; cd republish
$ pkgrecv -d . --raw -s http://openindiana.org/hipster/ package/pkg
$ cd package*    # package name contains a '/', and is url-encoded.
$ cd *           # we pulled down just the latest package by default
$ cat > fix-pkg
# change value of pkg.summary
<transform set name=pkg.summary -> edit value '.*' "IPS has lots of features">
# delete any signature actions
<transform signature -> drop>
# remove timestamp from fmri so we get our own
<transform set name=pkg.fmri -> edit value ":20.+" "">
^D
$ pkgmogrify manifest fix-pkg > new-manifest
$ pkgrepo create ./mypkg
$ pkgsend -s ./mypkg publish -d . new-manifest
```

## Example 2: Change Package Publisher

Another common use case is to republish packages under a new publisher name.

This can be useful, for example, when consolidating the packages from several different development teams' repositories into a single repository for integration testing.

Again, this can be achieved using the steps in *Example 1*, using `pkgrecv --raw`, running a `pkgmogrify(1)` transform on the resulting manifest, then republishing the transformed manifest.

A sample transform to change the publisher to "mypublisher" is:

```
<transform set name=pkg.fmri -> edit value pkg://[^/]+/ pkg://mypublisher/>
```

Iterating over all packages in the repository can be done with a simple shell script, that uses the output from `pkgrecv --newest` to process only the newest packages from the repository.

In the script below, we've saved the transform above in a file called `change-pub.mog`, and want to in republish from `development-repo` to a new repository `mypublisher`, changing the package publisher along the way:

```
#!/usr/bin/ksh93
pkgrepo create mypublisher
pkgrepo -s mypublisher set publisher/prefix=mypublisher
mkdir incoming
for package in $(pkgrecv -s ./development-repo --newest); do
    pkgrecv -s development-repo -d incoming --raw $package
done

for pdir in incoming/*/* ; do
    pkgmogrify $pdir/manifest change-pub.mog > $pdir/manifest.newpub
    pkgsend -s mypublisher publish -d $pdir $pdir/manifest.newpub
done
```

If necessary, we could modify this script to select only certain packages, make additional changes to the versioning scheme of the packages, and show progress as it republishes each package, for example.

# Appendix A

## Classifying Packages

The following are defined values for the package attribute `info.classification` with scheme `org.opensolaris.category.2008`, used by the Package Manager GUI to display possible packages. A typical entry as used in a package manifest might be:

```
set name=info.classification value=\
    "org.opensolaris.category.2008:System/Administration and Configuration"
```

Note that category and subcategory are separated by a "/". As usual, spaces in the attribute value require quoting.

Defined categories and subcategories for values are:

**Meta Packages**

- Builds
- Releases
- Developer Tools
- AMP Stack
- Office Tools

**Applications**

- Accessories
- Configuration and Preferences
- Games
- Graphics and Imaging
- Internet
- Office
- Panels and Applets
- Plug-ins and Run-times
- Sound and Video
- System Utilities
- Universal Access

**Desktop (GNOME)**

- Documentation
- File Managers
- Libraries
- Localizations
- Scripts
- Sessions
- Theming
- Trusted Extensions
- Window Managers

**Development**

- C
- C++
- Databases
- Distribution Tools
- Editors
- Fortran
- GNOME and GTK+
- GNU
- High Performance Computing
- Integrated Development Environments
- Java
- Objective C
- Observability
- Other Languages
- PHP
- Perl
- Python
- Ruby
- Source Code Management
- Suites
- System
- X11

**Drivers**

- Display
- Media

- Networking
- Other Peripherals
- Ports
- Storage

**System**

- Administration and Configuration
- Core
- Databases
- Enterprise Management
- File System
- Fonts
- Hardware
- Internationalization
- Libraries
- Localizations
- Media
- Multimedia Libraries
- Packaging
- Printing
- Security
- Services
- Shells
- Software Management
- Text Tools
- Trusted
- Virtualization
- X11

**Web Services**

- Application and Web Servers
- Communications

# Appendix B

## Converting SVR4 Packages to IPS

This appendix covers conversion of packages from SVR4 to IPS and highlights some aspects of the conversion that should be given special attention.

Chapter 4 goes into detail on how to package software in IPS. Developers with build environments that currently produce SVR4 packages should convert their build processes following the example in that chapter, rather than continuing to build SVR4 packages then converting those packages to IPS.

As with Chapter 4, the fundamental steps to packaging any software in IPS are:

- Generate a package manifest.
- Add necessary metadata to the generated manifest.
- Evaluate dependencies.
- Add any facets or actuators that are needed.
- Check the package with `pkglint(1)`.
- Publish the package.
- Test the package.

These steps remain essentially the same for SVR4 to IPS conversion and we will not repeat their explanations. There are a few steps that warrant more detailed explanations, and which are covered in this appendix.

In this appendix, a sample SVR4 package which is similar to the IPS package created in Chapter 4 is used.

### Generate a Package Manifest

`pkgsend generate` has support for scanning several different sources in order to generate manifests. In Chapter 4, we used a simple directory as the source. The `pkgsend(1)` utility can also read SVR4 packages, consulting the `pkgmap(4)` file in that package, rather than the directory inside the package that contains the files delivered.

While scanning the `prototype` file, `pkgsend(1)` also looks for entries that could cause problems when converting the package to IPS. The `pkgsend(1)` utility reports those problems and prints the generated manifest.

In this example, a SVR4 package will be used that has a `pkginfo` file:

```
VENDOR=My Software Inc.
HOTLINE=Please contact your local service provider
PKG=MSFTmypkg
ARCH=i386
DESC=A sample SVR4 package of My Sample Package
CATEGORY=system
NAME=My Sample Package
BASEDIR=/
VERSION=11.11,REV=2011.10.17.14.08
CLASSES=none manpage
PSTAMP=linn20111017132525
MSFT_DATA=Some extra package metadata
```

and a corresponding `prototype` file:

```
i pkginfo
i copyright
i postinstall
d none opt 0755 root bin
d none opt/mysoftware 0755 root bin
d none opt/mysoftware/lib 0755 root bin
f none opt/mysoftware/lib/mylib.so.1 0644 root bin
d none opt/mysoftware/bin 0755 root bin
f none opt/mysoftware/bin/mycmd 0755 root bin
d none opt/mysoftware/man 0755 root bin
d none opt/mysoftware/man/man1 0755 root bin
f none opt/mysoftware/man/man1/mycmd.1 0644 root bin
```

Running `pkgsend(1)` on the SVR4 package built using these files, the following IPS manifest is generated:

```
$ pkgsend generate ./MSFTmypkg | pkgfmt
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall

set name=pkg.summary value="My Sample Package"
set name=pkg.description value="A sample SVR4 package of My Sample Package"
set name=pkg.send.convert.msft-data value="Some extra package metadata"
dir  path=opt owner=root group=bin mode=0755
dir  path=opt/mysoftware owner=root group=bin mode=0755
dir  path=opt/mysoftware/bin owner=root group=bin mode=0755
file reloc/opt/mysoftware/bin/mycmd path=opt/mysoftware/bin/mycmd owner=root \
    group=bin mode=0755
dir  path=opt/mysoftware/lib owner=root group=bin mode=0755
file reloc/opt/mysoftware/lib/mylib.so.1 path=opt/mysoftware/lib/mylib.so.1 \
    owner=root group=bin mode=0644
dir  path=opt/mysoftware/man owner=root group=bin mode=0755
dir  path=opt/mysoftware/man/man1 owner=root group=bin mode=0755
file reloc/opt/mysoftware/man/man1/mycmd.1 \
    path=opt/mysoftware/man/man1/mycmd.1 owner=root group=bin mode=0644
legacy pkg=MSFTmypkg arch=i386 category=system \
    desc="A sample SVR4 package of My Sample Package" \
    hotline="Please contact your local service provider" \
    name="My Sample Package" vendor="My Software Inc." \
    version=11.11,REV=2011.10.17.14.08
license install/copyright license=MSFTmypkg.copyright
```

There are several points to note in the output above:

- The `pkg.summary` and `pkg.description` attributes were automatically created, using the data from the `pkginfo` file.
- A `set` action was generated from the extra parameter in `pkginfo` file. These are set beneath the `pkg.send.convert.*` namespace, with the intention that developers will use `pkgmogrify(1)` transforms to convert these to a more appropriate attribute name.
- A `legacy` action was generated, using data from the SVR4 pkginfo file.
- A `license` action was generated, pointing to the copyright file used in the SVR4 package.
- An error message was emitted pointing to a scripting operation that can't be converted.

Checking again, we can see a non-zero return code from `pkgsend generate`, and the error message again:

```
$ pkgsend generate  MSFTmypkg > /dev/null
pkgsend generate: ERROR: script present in MSFTmypkg: postinstall
$ echo $?
1
```

In this case, the package is using a `postinstall` script that can't be converted directly to an IPS equivalent. The script must be manually inspected.

This is the `postinstall` script in the package:

```
#!/usr/bin/sh
catman -M /opt/mysoftware/man
```

Its effects can easily be replaced by using a `restart_fmri` *actuator* pointing to an existing SMF service, `svc:/application/man-index:default` as described in Chapter 4. Also, see Chapter 9 for further discussion of actuators.

`pkgsend generate` will also check for the presence of class-action scripts and will produce error messages indicating which scripts should be examined.

It is impossible to give examples for every package scripting scenario that a developer can encounter when converting SVR4 packages to IPS packages. In IPS, the needed functionality probably can be implemented by using an existing action type or SMF service.

See Chapter 3 for details about the *action types* available, and Chapter 9 for a discussion on *actuators*.

## Verify the Package

We'll assume that any additional package metadata needed has been added to the manifest, and that dependency generation and resolution has been performed as per Chapter 4. Our next step is running `pkglint(1)` on the package.

A common source of errors when converting old SVR4 packages is mismatched attributes between directories delivered in the SVR4 package and those delivered by other packages on the system.

In this case, the directory action for `/opt` in the sample manifest has different attributes than those defined by the system packages.

Recall that in Chapter 3, we discussed the directory action, stating that all reference-counted actions must have the same attributes. When trying to install the version of `mypkg` that has been generated so far, an error will occur:

```
# pkg install mypkg
Creating Plan /
pkg install: The requested change to the system attempts to install multiple actions
for dir 'opt' with conflicting attributes:

    1 package delivers 'dir group=bin mode=0755 owner=root path=opt':
        pkg://mypublisher/mypkg@1.0,5.11-0:20111017T020042Z
    2 packages deliver 'dir group=sys mode=0755 owner=root path=opt':
        pkg://openindiana.org/developer/build/onbld@0.5.11,5.11-2018.0.0.18233:20190417T014131Z
        pkg://openindiana.org/SUNWcs@0.5.11,5.11-2018.0.0.18233:20190417T022040Z

These packages may not be installed together.  Any non-conflicting set may
be, or the packages must be corrected before they can be installed.
```

To catch the error before publishing the package, rather than at install-time, `pkglint(1)` can be used with a reference repository:

```
$ pkglint -c ./cache -r file:///scratch/oi-repo ./mypkg.mf.res
Lint engine setup...

PHASE                                          ITEMS
4                                          4292/4292
Starting lint run...

WARNING opensolaris.manifest001.1 Missing attribute 'org.opensolaris.consolidation' in pkg:/mypkg@1.0,5.11
ERROR pkglint.dupaction007        path opt is reference-counted but has different attributes across 3
duplicates: group: bin -> mypkg group: sys -> developer/build/onbld SUNWcs
```

In particular, notice the error message it produces about /opt having incorrect attributes. The extra `ldomsmanager` package that `pkglint(1)` reports was in the reference package repository, but was not installed on the test system, so it did not show up in the errors reported previously by `pkg install`.

## Other Considerations

While it is possible to install SVR4 packages directly on a system running IPS, we strongly recommend against this.

Apart from the `legacy` action, described in Chapter 3, there are no links between the two packaging systems, and they do not reference package metadata from each other.

IPS has commands such as `pkg verify` which can determine whether packaged content has been installed correctly. However if another packaging system legitimately installs packages, or runs install scripts that modify packaged files from IPS, errors might result.

Commands such as `pkg fix` or `pkg revert` could overwrite files that were delivered by a SVR4 package as well as an IPS package, potentially causing the packaged applications to malfunction.

Similarly, commands such as `pkg install`, which normally check for duplicate actions and common attributes on reference-counted actions, could also fail to detect potential errors when files from a different packaging system conflict.

With these pitfalls in mind, and given the comprehensive package development tool chain in IPS, developing IPS packages instead of SVR4 packages is recommended for OpenIndiana.