Software Distribution Legacy Guide

(Legacy)

Tools > Files & Software Installation

2006-07-24

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Software Distribution

Important: The information in this document is obsolete and should not be used for new development.

Software Distribution describes how to distribute your Mac OS X software and allow users to conveniently install it.

Important: This document has not been updated for Mac OS X v10.4. Most of the information is still accurate, but in some cases the behavior has changed, particularly with regard to the PackageMaker application. See Installer Release Notes for details on new features in Mac OS X v10.4.

Most Mac OS X software is distributed on CD or by Internet downloads, which are often provided as compressed disk images or in other archive formats. In the simplest case, you can set up your software so that users simply drag it from a CD or mounted disk image to the appropriate location on their hard disks. Apple recommends this type of installation whenever possible. For details and limitations, see "Drag-and-Drop Installation" (page 18). For additional information on working with disk images, see "Distributing Software With Internet-Enabled Disk Images" (page 21).

For software with more complex installation requirements, you can create an installation package with the PackageMaker application and let users install it with the Installer application. (Installer, located in `/Applications/Utilities`, is the native installer for Mac OS X. PackageMaker is installed with the Xcode tools and is located in `/Developer/Applications/Utilities`.) *Software Distribution*, along with the PackageMaker Help documentation, provides the information you'll need for this type of installation. For more information, see "Package-Based Installation" (page 19).

In some business and education environments, administrators can benefit by using a remote installation process. The Apple Remote Desktop application allows an administrator to simultaneously install software from a central location to multiple Macintosh computers, whether the software is prepared for "drag and drop" installation or is packaged by PackageMaker. For more information, see http://www.apple.com/remotedesktop/.

Although *Software Distribution* generally describes how to distribute and install applications, most of the information can be applied to frameworks, tools, kernel extensions, and other kinds of software. For a tutorial on how to install software with PackageMaker and Installer, including specific information on working with kernel extensions, see "Packaging Your KEXT for Distribution and Installation" in Darwin Documentation.

Who Should Read This Document

This document is intended for developers who need to distribute Mac OS X software. It assumes that you have some familiarity with the information about application packaging in *Bundle Programming Guide* in Core Foundation Resource Management.

Software Distribution does not describe installation for versions of the Mac OS prior to Mac OS X. The package-based techniques it describes are not intended for software installation on UNIX-based systems other than Mac OS X.

For specific limitations of the software, see "Limitations of PackageMaker and Installer" (page 40).

Organization of This Document

Software Distribution includes these articles:

- "Software Distribution Guidelines" (page 17) provides tips on the installation process, such as when to use drag-and-drop installation and when you might need to work with PackageMaker and Installer.
- "Distributing Software With Internet-Enabled Disk Images" (page 21) describes a mechanism for working with disk images that can provide an enhanced user experience in downloading files.
- "A Quick Look at PackageMaker and Installer" (page 25) describes terminology and concepts that will help you work effectively with packages.
- "About Packages" (page 29) describes the basic steps you typically use to create and test a package.
- "Installs and Upgrades" (page 33) defines the difference between an install and an upgrade and explains differences in how they are handled by Installer.
- "PackageMaker and Installer Features" (page 35) describes the major features available in PackageMaker and Installer. For some features, it points to other sections for more detailed information.
- "Creating a Package" (page 43) provides a detailed look at the steps involved in packaging your software.
- "The Installation Process" (page 53) lists the steps Installer takes during an installation, including a detailed list of the order in which executable files within a package are executed.
- "Customizing What Installer Shows to the User" (page 61) describes how you can customize parts of the Installer application's interface with the information and resources you build into your installation package.
- "Anatomy of a Package" (page 69) provides a detailed description of the contents of a package.
- "Anatomy of a Metapackage" (page 75) provides a detailed description of the contents of a metapackage.
- "Authorization, File Ownership, and Permissions" (page 79) describes the support in PackageMaker and Installer for requiring and performing authorization and for specifying file ownership and permissions.
- "Examining and Modifying Property Lists" (page 87) describes how to examine and directly modify the Installer keys that are used to store installation information in a package. It also provides reference for the Installer keys that you can add or modify.
- "Finding Previously Installed Software" (page 97) describes how you can you provide Installer with rules to help it find and upgrade software from a previous installation.
- "Specifying Version Information for Packaged Software" (page 107) describes how you can supply information to help Installer determine which files need to be replaced during an upgrade.
- "Specifying Installation Requirements" (page 111) describes how to insert information into the information property list for a package to request or require that certain hardware or software criteria be satisfied before the package is installed.

- "Checking the Installation With InstallationCheck" (page 125) describes how you can supply an executable file in your installation package that checks criteria you are interested in and disallows installation on systems that do not satisfy the criteria.
- "Checking the Installation Volume With VolumeCheck" (page 133) describes how you can supply an executable file in your installation package that checks volume criteria and disallows installation on volumes that do not satisfy the criteria.
- "Modifying an Installation With Scripts" (page 139) describes how you can supply script files (or other executables) that Installer executes at specified times during an installation.
- "Troubleshooting Packages and Installation" (page 145) provides tips for working with scripts, as well as a question-and-answer section on some common installation issues.
- "Topic Revision History" (page 151) describes changes made to this document.

Software Distribution Guidelines

This section provides a brief description of how to distribute Mac OS X software, and of the two primary mechanisms for installing it: drag-and-drop installation, which allows a user to simply drag the software to the appropriate folder; and package installation, which provides a more sophisticated set of installation options. It also lists benefits of working with PackageMaker and Installer, the native applications for package-based installation in Mac OS X.

Getting Software to the User

Whether your software can simply be dragged to the desired location or requires an installation process, you must first get the software to the user. Most software for Mac OS X is distributed on CD or by Internet downloads, which are often supplied as compressed disk images, or as archived files in various other formats.

For software that stores data or code in the resource fork of a file, the distribution mechanism must ensure that such data or code is not lost in transfer. This is especially likely to be a problem when files can be moved between different file systems. One solution is to distribute your software in an archived format (such as a Stuffit archive file) that prevents modification of the contents during transfer, and to have a user expand it on the target volume. This has the advantage of preserving data stored in resource forks—for related information, see "Preserving Resource Fork Data" (page 40).

System administrators in business and education environments may need to distribute and install site-licensed software from a central location. Apple Remote Desktop provides an ideal solution for this situation. For more information, see http://www.apple.com/remotedesktop/.

A **disk image** is a special kind of file that the Finder can mount as it would any disk. Mac OS X provides the Disk Utility application (located in `/Applications/Utilities`) for creating and mounting disk images. (Prior to Mac OS X version 10.3, there is also a separate Disk Copy application.) Disk image files typically have an extension of `img`, `dmg`, or `smi`. They can optionally be compressed, and can also be encrypted, requiring a password or phrase before mounting. For more information on working with disk images, use choose Help > Disk Utility Help in the DiskUtility application. For a brief tutorial on creating a disk image, see *Porting UNIX/Linux Applications to Mac OS X* in Darwin Documentation.

Note: You can also use the command-line tool `hdiutil` to work with disk images. For more information, see the man page for that tool.

Installing software from a downloaded disk image generally requires a number of steps before the actual installation can take place, followed by deleting the leftover files. Starting with Mac OS X version 10.2.3, the Disk Copy/Disk Utility application supports "internet-enabled" disk images, which can reduce the number of steps and improve the user experience. This mechanism is described in "Distributing Software With Internet-Enabled Disk Images" (page 21).

Note: You may want to use tools such as `tar` and `gzip` to distribute certain software, such as command-line tools. If so, refer to the man pages for those tools or to other UNIX documentation.

Drag-and-Drop Installation

Because most applications for Mac OS X are built as self-contained application bundles (defined below), users can install them in most situations just by dragging the bundle to a folder for which they have write permission. This drag-and-drop type of installation is the preferred method for installing Mac OS X software. For software of this type, you can provide simple installation instructions in a brief online or printed document.

One complication of using drag-and-drop installation is that it does not automatically support updating of prebinding. Prebinding is a process that can improve the user experience by providing a faster application launch. For more information on working with prebinding, see "Prebinding" (page 38).

An **application bundle** is a directory in the file system that stores executable code for an application and the software resources related to that code. An application bundle ends with the extension `.app` and is presented to the user by the Finder as a single file, with the extension hidden. Double-clicking the file launches the application executable, which is stored in a directory within the bundle (for example, see the MyApp executable in Listing 1). Bundles may even contain multiple executable files for different environments.

An application bundle can include localizations, and applications can automatically display a set of localized resources (including the application name itself) that matches a user's Languages preference.

Plug-ins and frameworks are also types of bundles, though frameworks are not presented by the Finder as a single file.

Note: You can examine the contents of a bundle in the Finder by Control-clicking it and choosing Show Package Contents from the contextual menu. In a Terminal window, you can display a bundle's contents with the `find` command. The Terminal application is available in `/Applications/Utilities`.

Listing 1 shows a partial layout for the application bundle for the application MyApp. Items preceded by a dash represent directories. The three directories `English.lproj`, `French.lproj`, and `Japanese.lproj` store language-specific resources for, respectively, U.S. English, French, and Japanese.

Listing 1 A partial bundle layout for a complex application

```
- MyApp.app
    - Contents
        - MacOS
            MyApp
          Info.plist
          PkgInfo
        - Resources
              Hand.tiff
              WaterSounds
            - English.lproj
                house.jpg
                InfoPlist.strings
                Localizable.strings
                CitySounds
```

```
- French.lproj
      house.jpg
      InfoPlist.strings
      Localizable.strings
      CitySounds
- Japanese.lproj
      house.jpg
      InfoPlist.strings
      Localizable.strings
      CitySounds
```

Because an application keeps everything it needs within its bundle, simple drag-and-drop installation reduces file-system clutter and eliminates dependencies on items residing elsewhere in the file system. It also gives users the option of not copying items to their hard disks that they are not particularly interested in (such as Read Me files). To uninstall an application, all users have to do is remove the application bundle from the volume.

Package-Based Installation

For some software, simple drag-and-drop installation is not sufficient. When the requirements for preparing an application for execution are more complex, you can put your software into a package, distribute it through any convenient means, and let users install it with an installer application.

This is the approach supported by PackageMaker and Installer, the native applications for package-based installation in Mac OS X. You use PackageMaker to create an installation package that contains all the information and resources required to install your software. When a user double-clicks the package file, Installer (included in every Mac OS X installation) is launched to handle the actual installation.

Note: This document does not describe third-party or open-source packaging and installation mechanisms.

"Reasons for Using PackageMaker and Installer" (page 19) lists some of the advantages of this type of installation. Information on using PackageMaker and Installer is found throughout this document—for an introduction, see "A Quick Look at PackageMaker and Installer" (page 25).

Reasons for Using PackageMaker and Installer

At the highest level, using PackageMaker and Installer to install your software provides these options:

- a professional-looking presentation, customized to your particular software
- an installation process that is easy for unsophisticated users
- support for upgrading your software, which may require replacing only certain files
- support for custom installation (so that a user can choose to install part or all of the packaged software)
- the ability to ensure correct prebinding (a mechanism to provide faster application launching, described in "Prebinding" (page 38))

On a more detailed level, PackageMaker and Installer provide fine control over the installation process, including

- the ability to provide a Welcome page, Read Me page, and custom background image
- the ability to perform operations before installing, such as
  - requiring the user to agree to a license
  - requiring the user to authenticate before installing in a privileged location
  - checking for the presence of required hardware or other features
- control over details such as whether an installation
  - allows the user to specify a destination for the software or requires a specific destination
  - recommends or requires a restart, logout, or shutdown after completion
  - uses the file ownership and permissions of the user installing the software or of the packaged software itself

For a detailed description of the available features, see "PackageMaker and Installer Features" (page 35), as well as PackageMaker Help.

Distributing Software With Internet-Enabled Disk Images

Disk images have become the preferred transport mechanism for downloading files in Mac OS X. Starting with Mac OS X version 10.2.3, the Disk Copy application (located in `/Applications/Utilities`) supports "internet-enabled" disk images, which help create a better user experience when installing from disk images. This section describes that process and how you can take advantage of it.

Note: Starting in Mac OS X version 10.3, the features of the Disk Copy application were integrated into the Disk Utility application, also located in `/Applications/Utilities`.

Improving the User Experience

Currently, the typical user-experience for Mac OS X software installation looks something like the following:

1. The user uses a browser to download a disk image file (a file with a `.dmg` extension) that has been archived in Stuffit or MacBinary format.
2. The browser launches StuffIt Expander to expand the downloaded file. The user watches a progress bar.
3. StuffIt Expander launches Disk Copy to mount the disk image. The user watches another progress bar.
4. The user locates the mounted image, opens a Finder window (if one isn't set to open automatically), and copies the contents out of the disk image.
5. The user unmounts the disk image.
6. The user throws away the `.dmg` file.
7. The user throws away the compressed `.dmg` file, if StuffIt Expander is not configured to delete compressed files automatically.

However, it is now possible to eliminate many of these steps. Because read-only `.dmg` files use a flat file format and Disk Copy can create compressed disk images, it is unnecessary to convert a `.dmg` file to either StuffIt format or MacBinary format. As a result, steps 2 and 7 can be eliminated. New internet-enabled disk images eliminate most of the remaining steps, and the user experience is simplified as follows:

1. The user downloads a `.dmg` file using a browser.
2. The browser launches Disk Copy with the `.dmg` file. The user watches a progress bar while Disk Copy mounts the disk image, copies the contents out of the image, unmounts the image, and moves the `.dmg` file to the Trash.
3. The user locates the downloaded files and perhaps moves them to their final destination.

In the new model, the user never needs to handle or see the `.dmg` file or the mounted disk image. Instead, the user sees only the files contained within the disk image. In other words, the `.dmg` file becomes an implementation detail that is transparent and irrelevant to the user; the included files appear at the download location as if they were downloaded directly.

When it moves the `.dmg` file to the trash, Disk Copy clears the internet-enabled flag, so that if the user decides to remove the file from the trash for future use, it will operate as a standard disk image.

Note: If necessary, you can use PackageMaker to package your software, then distribute the package via internet-enabled disk image. After receiving the software, the user must install it with Installer (and delete the package file, if they don't want to keep it around). For information on when you might want to do this, see "Reasons for Using PackageMaker and Installer" (page 19).

Creating An Internet-Enabled Disk Image

An internet-enabled disk image is identical to a regular disk image except that it has a special flag set. You create the disk image normally and then set the internet-enabled flag with the following command, executed in a Terminal window, where `<pathToDmg>` is the path to the image file:

```
hdiutil internet-enable -yes|-no|-query <pathToDMG>
```

Use `-yes` to enable the flag, `-no` to disable the flag, and `-query` to test whether the `.dmg` file is already internet-enabled.

You can set the internet-enabled flag only for read-only disk images. Read-write disk images cannot be internet enabled. Older `.img` and `.smi` files also cannot be internet enabled.

Disk Copy clears the internet-enabled flag after it processes a disk image. For this reason, if you need to test the file before putting it online, you must make a copy of the disk image first and test the copy. You cannot retrieve a processed disk image from the Trash and then place it online—it will no longer be internet-enabled.

The internet-enabled flag does not interfere with previous versions of Disk Copy. Previous versions of Disk Copy will simply ignore the flag and treat the disk image as it would a non-internet-enabled disk image.

Note: For a brief tutorial on creating a disk image, see *Porting UNIX/Linux Applications to Mac OS X* in Darwin Documentation.

Adding a License Agreement to a Disk Image

The Disk Copy application has the ability to display a multi-lingual software license agreement before mounting a disk image. The image will not be mounted unless the user indicates agreement with the license. You can obtain a software development kit (Software License Agreements for UDIFs) that explains how to add license agreements to Mac OS X-native UDIF disk images at http://developer.apple.com/sdk/index.html.

How Disk Copy Handles an Internet-Enabled Disk Image

Disk Copy processes an internet-enabled disk image a little differently than a regular disk image, although the differences are mostly additional behavior. The disk image can still contain a software license agreement or it can be encrypted and require a password. The following are the new and modified Disk Copy behaviors when handling an internet-enabled disk image:

- Disk Copy mounts the disk image in a hidden location so that it is not displayed on the Desktop or in Finder windows.
- After mounting the image, Disk Copy copies the contents of the disk image out of the image and into the same directory in which the `.dmg` file is stored, which is usually the user's download folder. If there is only one file visible to the user in the disk image, that file is copied directly to the destination. Any hidden files, such as a `.DS_Store` file or a folder background image, are not copied. If there are multiple files visible to the user at the root of the disk image, a new directory is created at the destination and the visible files are copied into it. If present, the disk image's `.DS_Store` file is also copied to the new directory, thereby preserving icon positions; other hidden files are not copied. The new directory's name is the same as the disk image's, but without the `.dmg` extension.
- After copying the disk image's contents, Disk Copy unmounts the image, clears the internet-enabled flag in the `.dmg` file, and moves the `.dmg` file into the Trash. Disk Copy does not delete the `.dmg` file. If the user wants to keep the `.dmg` file of a download, perhaps for archival purposes, the user can retrieve the file from the Trash. Because Disk Copy has cleared the internet-enabled flag, however, the `.dmg` file now behaves like a regular disk image file. In other words, the internet-enabled behavior occurs only when the disk image is initially downloaded.

Caveats for Internet-Enabled Disk Images

Although supplying an internet-enabled disk image can reduce the number of steps a user has to perform to install your software, there are some issues you should be aware of:

- To ensure optimum launch time for your application, be sure you address possible prebinding issues described in "Drag-and-Drop Installation" (page 18) and "Prebinding" (page 38). In brief, you should provide users with information on why they might need to update prebinding for your application and how to to so.
- Some users may be confused when they double-click a disk image file and it then "goes away" (to the Trash).
- A user who finds and retrieves the image file from the Trash may again be confused that double-clicking it a second time doesn't produce the same behavior.
- Some users may not be experienced with the use of disk images, whether these images are internet enabled or otherwise.

If you are concerned about any of these issues, you can provide instruction at your download site or through a Read Me file or other mechanism. You can tailor the information you provide to the sophistication of your target audience.

A Quick Look at PackageMaker and Installer

This section provides a brief introduction to PackageMaker and Installer, as well as an introductory look at creating a package.

About PackageMaker and Installer

You use the PackageMaker application to create package files that contain all the information and resources required to install your software. In some cases, you supply information through the PackageMaker user interface. For example, you use the interface to supply the title and description for the package and the location of the software to be placed in the package. In other cases, you prepare information before running PackageMaker. For example, you might create files such as a Read Me file or license agreement for your software.

A package has the file extension `.pkg` and contains both the software to install and various resources used in the installation process. Double-clicking a package file launches Installer, which installs your software according to the information you have supplied and the user's responses. Installer then processes the package (including displaying a license agreement and other information, and executing scripts if the package contains any).

You can also create metapackages, which have the file extension `.mpkg` and contain other packages or metapackages. Metapackages allow a user to do a custom installation, where they can choose which parts of the software to install. Both packages and metapackages are folders that are presented to the user as a single file.

PackageMaker also creates package definition files, with extension `.pmsp`, which store the information needed to create a package. You can open a definition file, make changes in PackageMaker, then recreate the associated package file. Metapackage definition files, with extension `.pmsm`, serve a similar function for creating and modifying metapackages.

The default process provided by PackageMaker and Installer can handle most packaging and installation requirements, but when special handling is required, you can supply scripts that are executed at well-defined times during the installation.

A Quick Look at Creating a Package

To package your software with PackageMaker, you typically use steps like the following (a more detailed version is provided in "Creating a Package" (page 43)):

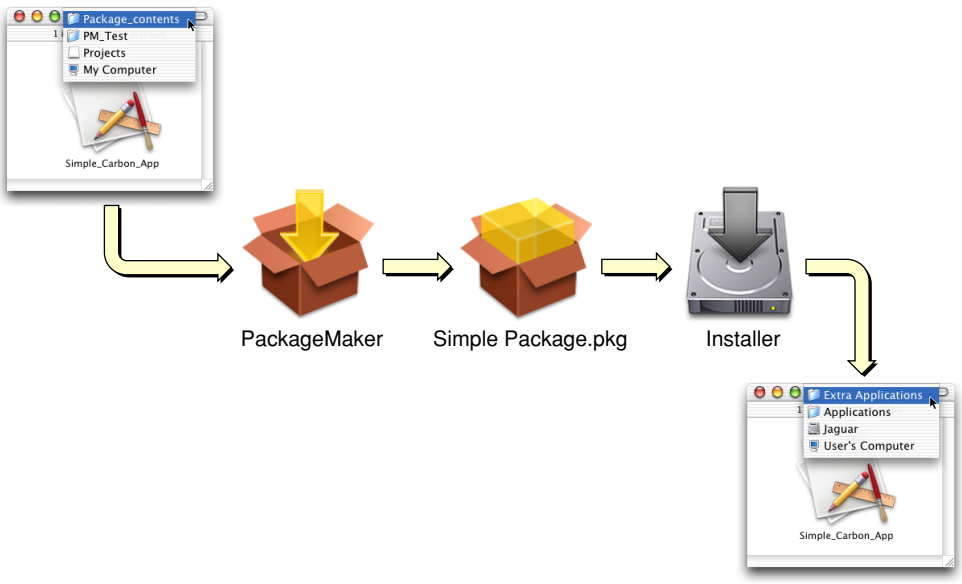
1. Create your software.
2. Create additional resources that may be needed as part of the installation, such as a Read Me file or a license agreement.

3. Store the software and resources in a suitable directory structure.
4. In PackageMaker, create a package file and a package definition file:
    a. Supply information such as the location of your software and resources, and how the software should be installed (whether it requires authorization, whether it must be installed at a specific destination, whether it requires a restart after installation, and so on).
    b. Use File > Create Package to create a package file containing the information needed to install the software. The package file has the extension `.pkg`.
    c. Use File > Save to save a PackageMaker package definition file as a template to recreate the package later, if necessary. The package definition file has the extension `.pmsp`.

    These file types are described in "File Types Created by PackageMaker" (page 30).
5. In the Finder, double-click the package to launch Installer and test the installation.
6. To make changes, either double-click the package definition file and go to step 4 or, in some cases, modify the contents of the package directly (for information, see "Examining and Modifying Property Lists" (page 87)) and go to step 5.
7. When the package is fully tested, distribute it to users for installation.

Figure 1 shows a simplified version of this process, taking an application from its location on your disk, to its final installation in the `/Applications` directory on a user's volume. Though this process may appear simple, it provides a multitude of features and gives you a great deal of control over the installation process.

Figure 1 Creating a package and installing software



You can provide useful installation packages using just the default features provided by PackageMaker and Installer. For example, Figure 2 shows the Installer default Introduction pane for a package that installs a simple Carbon application. However, with a little more effort, you can add resources to customize the installation with a Read Me, a welcome message, a license agreement, and a background image, as described in "Customizing What Installer Shows to the User" (page 61).

While these features should be sufficient for the majority of installations, your software may require more complex handling. If so, it can provide it through script files that are executed at specific times during the installation. These scripts, or other executables, can check for required hardware, determine which volumes are suitable for installation, and more. For more information, see "Customizing an Installation With Executable Scripts" (page 37).

Figure 2 The Installer default Introduction pane

In some cases, you may want to edit the contents of a package file directly. For information, see "Examining and Modifying Property Lists" (page 87).

About Packages

Packages are a key part of distributing and installing software with PackageMaker and Installer. The terminology and concepts described in this section will help you work effectively with packages.

What Is a Package?

The word "package" can have a number of different meanings. This document uses the term **installation package** (or simply **package**) to refer to the package files created by PackageMaker. These files have the following properties:

- They have the extension `.pkg`.
- They are presented by the Finder as a single file (though you can examine their contents by Control-clicking on the package and choosing Show Package Contents from the contextual menu, or by using the `find` command in a Terminal window).
- They contain various files and directories in well-defined locations within the package
- The files and directories in a package together provide the information needed by the Installer application to present the packaged software for installation.
- Double-clicking a package launches the Installer application.

During installation, Installer creates another package file (with extension `.pkg`) to serve as an installation receipt, and stores it in `/Library/Receipts`. This document uses the term **installation receipt** (or simply **receipt**) to refer to this kind of package.

For more information on receipts, see "Installs and Upgrades" (page 33). For more information on the file types created by PackageMaker, see Table 1 (page 30). For information on when you might want to edit the contents of a package directly, see "The Package as a Black Box" (page 32).

What Is a Metapackage?

A metapackage is a file that includes a list of packages (and possibly other metapackages) and any additional information needed to install them; the packages themselves, however, are not actually contained in the metapackage. Double-clicking a metapackage launches Installer. During installation for a metapackage, at the point where a user can click a button to install or upgrade the software, they also have the option of clicking a Customize button to perform a custom install.

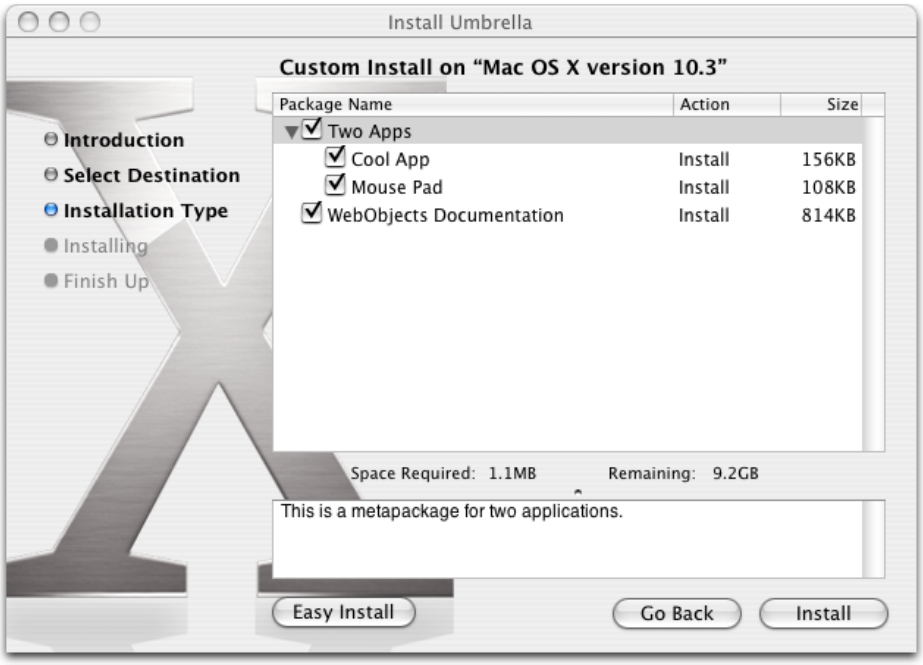
The Custom Install pane lets the user choose from the packages contained in the metapackage (and in any metapackages it may contain). The user can also click an Easy Install button to go back to the Easy Install pane. The package creator has the option of making any package (or metapackage) in a metapackage be required, in which case the user cannot deselect that package (or metapackage). Figure 1 shows the Custom

Install pane for a metapackage that is being installed on a volume named "Mac OS X version 10.3". The metapackage contains one package (WebObjects Documentation) and one metapackage (Two Apps), which in turn contains two packages (Cool App and Mouse Pad). In this example, neither the metapackage nor any of the packages is required.

It's a good idea to give users as much flexibility as possible, so you should generally create packages for any component of your software that a user might reasonably wish to install separately, then combine them in a metapackage.

For additional details on working with metapackages, see "Installs and Upgrades" (page 33), "Anatomy of a Metapackage" (page 75), and "Execution Order for a Complex Metapackage" (page 56).

Figure 1 A custom install for a metapackage containing one package and one metapackage



File Types Created by PackageMaker

Table 1 lists the kinds of files PackageMaker can create. Each of these files has a specific file extension and serves a specific purpose, as described in the table.

Table 1 File types supported by PackageMaker and Installer

File type	Extension	Purpose
package	`.pkg`	Encapsulates software and any additional information needed to install it. Double-clicking launches Installer and starts installation. For more information, see "Anatomy of a Package" (page 69).

File type	Extension	Purpose
package definition	`.pmsp`	Stores information needed to create a package file. Double-clicking launches PackageMaker. As you build and test a package file, you make changes to the corresponding package definition file.
metapackage	`.mpkg`	Encapsulates a list of packages (and possibly metapackages) and any additional information needed to install them. Double-clicking launches Installer. For more information, see "What Is a Metapackage?" (page 29).
metapackage definition	`.pmsm`	Stores information needed to create a metapackage file. Double-clicking launches PackageMaker.

A Simple Package

A package is a directory, presented by the Finder as a single file, that contains all the information the Mac OS X Installer application needs to install your software. That includes the software itself, as well as files that are used only during the installation process. The software to be installed is typically stored a compressed archive, and may include application bundles, frameworks, documentation, and other kinds of files. The files used only as part of the installation process can include localized files for customizing the installation, scripts for checking the installation environment or performing other operations, and property list files created by PackageMaker to store various information about the installation.

Note: You can also create or modify a package's property list files directly. For more information, see "Examining and Modifying Property Lists" (page 87).

Figure 2 shows the contents of a very simple installation package file.

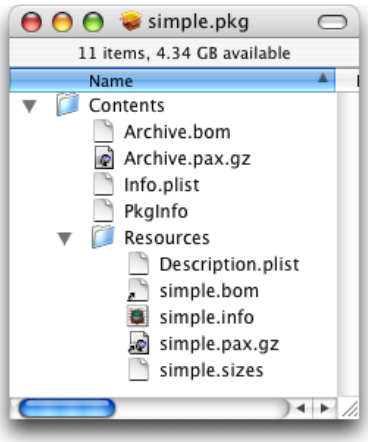
Important: Packages created with the version of PackageMaker distributed with Mac OS X version 10.2.0 or later have the format shown in Table 1, and are compatible with versions of Installer distributed with earlier versions of Mac OS X.

To display the contents of an installation package (or any other directory displayed in the Finder as a single file), Control-click the file and choose Show Package Contents from the contextual menu.

Note: You can display the contents of a package in a Terminal window with the `find` command. For example, for a package named `MyApplication.pkg` in the current directory, type `find MyApplication.pkg`.

For a detailed description of the files in an installation package file, see "Anatomy of a Package" (page 69).

Figure 2 The contents of a simple installation package



The Package as a Black Box

Some developers may be able to treat a package as a "black box," created by PackageMaker and modified only through PackageMaker. In this scenario, you save a corresponding package definition (`.pmsp`) file for each package file you create and, if you need to change the package, you open the package definition file in PackageMaker, use the application user interface to make changes, then create a new package file.

However, you may find it convenient to modify the contents of a package file directly. For example, it can be faster to test and modify a package's script files in the Finder or in a shell than to edit and resave the package in PackageMaker each time you modify a script. Or you might want to modify the `Resources` directory directly to test localized resources.

Additionally, information for some features cannot currently be supplied through the PackageMaker user interface, so if you need those features, you must modify the package's `Info.plist` or `Description.plist` files directly. A primary example of this is the feature, added in Mac OS X version 10.3, which allows you to specify requirements for your installation package by editing the package's property list. For more information, see "Examining and Modifying Property Lists" (page 87) and "Specifying Installation Requirements" (page 111).

Some developers may also choose to modify package files directly as part of automating the build process or automating the process of distributing software. Those topics are not described in this document.

Finally, you may choose to modify packages directly as a convenience, rather than having to go back through PackageMaker each time you need to make a change. If so, be sure you consider some of the potential issues described in "Troubleshooting Packages and Installation" (page 145).

Installs and Upgrades

During installation, Installer creates a package file with the same name as the package being installed to serve as an installation receipt. The receipt serves as a record for an installation and contains the same information that was in the installation package, except for the archive of the software to be installed. That information includes a `.bom` file, which contains, in binary format, a list of the files that were actually installed. Installer stores the receipt in `/Library/Receipts` on the installation volume, creating the directory if it does not already exist.

Installer uses the presence or absence of a receipt on the installation volume to determine whether an installation is an install or an upgrade. That is, Installer assumes software is being installed for the first time if there is no receipt with the same name as the package being installed. If there is such a receipt, the installation is an upgrade.

How Installer Handles Installs and Upgrades

An installation, or install, is treated differently than an upgrade in the following ways:

1. Installer shows the appropriate text ("Install" versus "Upgrade") to the user.
2. For an upgrade, if the package supplies appropriate version information, Installer may need to replace only certain outdated files, and may remove previously installed files that are not part of the current installation.

   For details on how Installer determines which files to install and remove during an upgrade, see "Specifying Version Information for Packaged Software" (page 107). And for related information, see "Finding Previously Installed Software" (page 97).
3. Depending on the contents of the package, Installer may execute different scripts as part of the installation. For example, for an install it will execute `preinstall` and `postinstall` scripts, if present; for an upgrade, it will execute `preupgrade` and `postupgrade` scripts, if present. (It always executes `preflight` and `postflight` scripts, if present.) These scripts are described in "Modifying an Installation With Scripts" (page 139).

Important: Installer does not currently support uninstalling software, although during an upgrade, it will remove previously installed files that are not part of the current installation.

That is, after the Installer has determined an upgrade is being performed, it will compare the names of files in the previous install (from the receipt) against the names of files currently chosen to be installed and remove any files not in the new install.

Installs and Upgrades With Metapackages

When a user installs packages from a metapackage, Installer stores a receipt for each package the user installs, but not for the metapackage itself. In the Custom Install pane for a metapackage, Installer will show which packages are installs and which are upgrades, based on the presence or absence of receipts for the individual packages.

A metapackage itself is considered an install only if all of the packages are installs; if any package is an upgrade then the metapackage is also considered an upgrade (and the button is labeled Install or Upgrade accordingly). The install or upgrade status of the metapackage determines which preinstall, preupgrade, postinstall, or postupgrade scripts in the metapackage are executed (if any are present). Preinstall and preupgrade scripts for a metapackage are run before any package is installed; postinstall and postupgrade scripts are run after all packages have been installed.

Install and upgrade scripts for an individual package in a metapackage are executed according to the install/upgrade status of the package. Preinstall and preupgrade scripts for an individual package are run before the package is installed; postinstall and postupgrade scripts are run afterwards.

For more information on which scripts are executed and when, see "Execution Order for a Complex Metapackage" (page 56).

PackageMaker and Installer Features

This section describes the key features provided by PackageMaker and Installer. For some features, it points to more detailed information in separate sections.

Before reading this section, you should be familiar with the information in "About Packages" (page 29).

For basic information on using PackageMaker, and on items in its user interface, see PackageMaker Help and the application's help tags.

Note: You can use PackageMaker from the command line. For information on how to do so, see the man page for the `packagemaker` tool. You can view the man page (available starting in Mac OS X version 10.3) by typing `man packagemaker` in a Terminal window; you can also view man pages in Xcode with Help > Open man page…

Authorization and Permissions

Installing a package may result in actions that require root or administrator privileges. PackageMaker allows you to specify whether root, administrator, or no authorization is required. Installer will then perform authorization if the user performing the installation lacks the required privileges. When authorization is required, Installer displays an authentication dialog as the first step of the installation.

It is important to ensure that the files that Installer places on a user's computer, and the directories they reside in, have the appropriate owner and permissions. For a detailed description of this issue, see "Authorization, File Ownership, and Permissions" (page 79).

Customizing the Installer Interface

You can customize parts of the Installer application's interface with the information and resources you build into your installation package with PackageMaker. For example, you can optionally supply localizable files that provide text for the following Installer panes:

- Introduction (or Welcome)
- Read Me
- License

In addition, you can supply an image that is displayed in the background of the installer window. You can specify how Installer scales and positions the image, and you can supply localized versions of the image. And finally, you must provide a title and you should also provide a description for your installation package, and can provide similar information for a metapackage.

For a detailed description of this feature, see "Customizing What Installer Shows to the User" (page 61).

Allowing Users to do Custom Installations

PackageMaker and Installer support custom installations, in which a user can choose which individual software items to install. To provide a custom installation, you create a separate package for each separately installable part of your software, then combine the packages in a metapackage. A metapackage is a file that encapsulates a list of packages (and possibly other metapackages) and any additional information needed to install them.

So, for example, if your software includes an application and some optional fonts, the suggested method of packaging and distribution is to package the fonts in one package, the application in another, and distribute them as a metapackage.

For more information on custom installs, see "What Is a Metapackage?" (page 29) and "Anatomy of a Metapackage" (page 75).

Specifying Where Software Will Be Installed

PackageMaker and Installer provide some flexibility in specifying a destination for installing software. In working with a single package, you can

- specify a default installation directory
- arrange various components of the software to be installed in various subdirectories located off the installation directory
- require that the software be installed only on the currently booted volume; if not, the user can choose any mounted volume; but see also "Customizing an Installation With Executable Scripts" (page 37) for information on checking the installation volume
- specify that the software is relocatable, which allows the user to choose a destination

When you combine individual packages in a metapackage, the user will be able to choose a destination if any of the individual packages is relocatable. If this is the case, the user will be able to choose only one destination for all relocatable packages they install, not separate destinations for each relocatable package; installed packages that are not relocatable will not be relocated. Note, however, that prior to Mac OS X version 10.3.4, relocation did not work reliably for metapackages.

For a detailed description of this feature, see "Specify the Installation Destination" (page 47).

Taking Additional Control of an Installation

PackageMaker and Installer support two mechanisms for providing additional control over an installation, described in the next sections.

Specifying Installation Requirements in a Property List

Starting in Mac OS X version 10.3, you can specify installation requirements by adding key-value pairs to the package's information property list. You can use this mechanism to do many of the same tasks described in "Customizing an Installation With Executable Scripts" (page 37), and should make use of this mechanism where needed.

For a detailed description of this feature, see "Specifying Installation Requirements" (page 111). For more information on editing property lists, see "Examining and Modifying Property Lists" (page 87).

Customizing an Installation With Executable Scripts

PackageMaker and Installer allow you to provide scripts (or other executable files) to provide additional control over an installation. You place scripts with the specified names in the `Resources` directory in your package file, or in some cases in a localized directory within the `Resources` directory. Installer executes these files, which must have their permissions set to be both executable and readable, at well-defined times.

Installation Checking

Before proceeding with an installation, you can test whether a user's system meets certain criteria, such as minimum hardware requirements or the presence of required files or directories. To do so, you supply an executable file named `InstallationCheck` in your installation package. This file checks the criteria you are interested in and cancels the installation, if necessary.

For a detailed description of this feature, see "Checking the Installation With InstallationCheck" (page 125).

Volume Checking

You can prevent users from installing your software on volumes that do not meet certain criteria. To do so, you supply an executable file named `VolumeCheck` in your installation package that determines whether a volume should be enabled or disabled as a possible installation destination.

For a detailed description of this feature, see "Checking the Installation Volume With VolumeCheck" (page 133).

Executing Scripts Before, During, and After Installation

You may need to configure the destination environment before installation or to perform additional processing after installation. The Installer application supports these capabilities by looking for certain named executable files in the installation package and, if found, executing them at specified times during installation.

For a detailed description of this feature, see "Modifying an Installation With Scripts" (page 139).

Localization

For many of the customizable features provided by PackageMaker and Installer, you can localize the information you provide:

- To localize welcome, Read Me, and license files, see "Localizing the Welcome, Read Me, and License Files" (page 63).
- To localize background images, see "Additional Tips for Background Images" (page 68).
- To localize messages shown when a user tries to install on a system that you have identified as inappropriate for an installation, see "File Placement for Installation Checking" (page 126).
- To localize messages shown when a user tries to select a volume that you have identified as inappropriate for an installation, see "File Placement for Volume Checking" (page 134).

For information on the localized folder names you can employ in packages you create, see "Localized Folder Names" (page 70).

Prebinding

Prebinding is the process of computing at build time the addresses for the symbols imported by libraries and applications, so that less work needs to be performed by the dynamic linker at runtime. Prebinding can improve the user experience by providing a faster launch, and is recommended for all applications.

If a user installs your software on a system with a set of libraries that is different than those present when the application was built, the prebinding information cannot be used and application launch is slower. Installer automatically attempts to update prebinding information for the software it installs by running the `update_prebinding` tool. If your software uses drag-and-drop installation, you can provide instructions for how to run this tool through a Read Me file, printed documentation, or other mechanism.

Prebinding is applicable only for Mach-O executables. (Mach-O is the native binary format for executable files in Mac OS X and is the only format supported by Project Builder.) In addition, there are some circumstances where prebinding information cannot be updated or used. For details, see the following:

- "Prebinding Your Application"
- The man pages for `update_prebinding` and `redo_prebinding`.

Helping Installer Find Previously Installed Software

The Find File feature allows you to provide Installer with rules to help it find and upgrade software from a previous installation. This feature is most useful for software that can be easily moved by the user after installation, such as application bundles.

For a detailed description of this feature, including new capabilities added in Mac OS X version 10.3, see "Finding Previously Installed Software" (page 97).

Restarting, Logging Out, or Shutting Down

PackageMaker provides several options for suggesting or requiring a restart, logout, or shutdown after installation. Note that these options should seldom be required—for example, you might only need a restart after installing system software, such as a kernel extension.

The options available in PackageMaker are:

- No Restart Required
- Recommended Restart
- Required Restart
- Shutdown Required

Important: Restart actions are effective only when the user installs on the currently booted volume. Otherwise, no action takes place.

Logging out is an additional restart option that, as of Mac OS X version 10.3, is not available through the PackageMaker interface, but can be set directly by modifying a package's information property list. See "Other Keys You Might Want to Modify" (page 96) for details.

There currently is no indication of a restart action until the user actually starts the installation. At that point, Installer displays a message warning the user that installing the software requires them to restart their computer, and allows them to cancel the installation. Installer currently displays the same message for all restart actions.

The difference between a Recommended Restart and a Required Restart or Shutdown is that the Quit item in the Installer menu is enabled for a Recommended Restart.

Error Reporting

As you create packages, test them, and make modifications, you need feedback about just what Installer is doing. If you supply scripts to customize your installation, you'll need to know what the scripts are doing as well.

Once you've launched Installer, you can choose File > Show Log to open the Installer Log window. This window displays the output from the scripts described in "Modifying an Installation With Scripts" (page 139), such as a preinstall or preupgrade script.

If you've supplied one of the Installation Check or Volume Check scripts described in "Checking the Installation With InstallationCheck" (page 125) or "Checking the Installation Volume With VolumeCheck" (page 133), you can view script output in the Console application (located in `/Applications/Utilities`).

For additional information on debugging packages, see "Troubleshooting Packages and Installation" (page 145).

Preserving Resource Fork Data

Prior to Mac OS X and Carbon, application resources were stored in the resource fork of the application executable. In Mac OS X and for Carbon applications generally, resources should be put in the data fork of a separate resource file, not the resource fork of the executable. The primary reason for moving application resources out of resource forks is to enable applications to be seamlessly moved around different file systems without loss of their resources; this would include transfer mechanisms such as BSD commands, FTP, email, and Windows and DOS copy commands. Most other computing environments, including the Web, recognize single-fork files only, and tend to delete the resource fork of HFS and HFS+ files.

When you package software with PackageMaker, you supply a location from which to gather the files to be packaged. If any of these files (such as a Classic application) has a resource fork, PackageMaker supports splitting the file before packaging. Such split files will be reassembled by Installer when the package is installed. Although splitting files with resource forks is optional, if you do not split them, the resource fork data will be lost and the file will be unusable.

Because resource fork splitting changes the files that are split, it is recommended that you operate on copies of these files while creating packages.

Support for preserving resource fork data was added to PackageMaker and Installer for Mac OS X version 10.2.

Limitations of PackageMaker and Installer

The following are some issues and limitations to be aware of. For some specific examples of possible issues, see "Troubleshooting Packages and Installation" (page 145).

- Installer does not currently support uninstalling software.
- Installer runs only in Mac OS X, although it can be used to install software for previous versions of the Mac OS (to locations that are accessible when it is running).
- Prior to Mac OS X version 10.2, PackageMaker did not attempt to preserve data stored in the resource fork of a file.
- When adding localized information to packages, you can use the folder names described in "Localized Folder Names" (page 70). However, if you construct other folder names from the ISO 639 standard for language codes and the ISO 3166 standard for country codes, they may not be recognized by Installer when installing your package.
- Regarding backward and forward compatibility of packages:
  - Packages created with versions of PackageMaker prior to the one distributed with Mac OS X version 10.2 (Jaguar) work with any version of the operating system.
  - Packages created with the version of PackageMaker that shipped with the Jaguar Developer Tools in 2002 work only with Mac OS X version 10.2 and later.
  - In most cases, packages created with PackageMaker 1.1.10 (which first shipped with the December 2002 Developer Tools CD) and later work with any version of Mac OS X. However, see "Why does my package fail in Mac OS X version 10.1 but not in version 10.2?" (page 146).

- If you do not use care, it is possible to create serious permissions problems for users installing your software. It is also possible to modify a user's system by replacing symbolic links with actual directories. For more information, see "Authorization, File Ownership, and Permissions" (page 79).
- Installer supports some features that are not currently accessible through the PackageMaker user interface. For more information, see "Editing a Property List" (page 88).
- The following features in the PackageMaker user interface are not supported:
  - On the Info pane for a package or a metapackage, information in the Version and Delete Warning fields is not used.
- An application that is distributed on multiple CDs cannot be installed from one package. If your software requires multiple CDs, break it up into smaller packages and provide installation instructions on how to install it.
- Prior to Mac OS X version 10.3.4, relocation did not work reliably for metapackages. That is, if a metapackage contained one or more relocatable packages, and the user chose an install location, the packages would not necessarily be installed in the correct location.

Creating a Package

This section describes the general steps you take to package your software. It assumes you are familiar with the information in "About Packages" (page 29), which introduces packages and describes the contents of a simple package. You should also be familiar with the information in "PackageMaker and Installer Features" (page 35).

This section is not strictly a tutorial. For a step-by-step tutorial that describes how to create a package, see *Packaging Your KEXT for Distribution and Installation*, in Darwin Documentation. That document describes how to create a package for installing a kernel extension (or KEXT), but most of the steps are common to packaging any kind of software.

In addition, you can read about many of the steps for creating a package in PackageMaker Help. You can also learn about the PackageMaker user interface through the available help tags.

Preparation

Before you start packaging your software, it is worthwhile to consider your installation goals and make some decisions. The following issues apply to most software installations:

- Do you really need to use an installation process, or can you allow users to simply drag your software to an appropriate location, as described in "Drag-and-Drop Installation" (page 18)?
- How will you be distributing your software? If you're distributing software on a CD, you can be less sensitive to size issues. However, if you're distributing software through Internet downloads, you may want to take steps to minimize the size of your package:
  - Be sure you choose to compress the package archive (this is the default setting).
  - If possible, consider providing your software in separate packages.
  - Avoid steps that can make the package unnecessarily large, such as using a large background image and providing localized copies for many languages.
- Are some parts of your software optional? If so, consider distributing your software with a metapackage, which allows the user to do a custom install.

  For more information, see "What Is a Metapackage?" (page 29).
- Does your software need to reside in a specific location, or is it relocatable (the user can choose a location for it)? If your software must be installed in more than one location (such as `/Applications` and `/System/Library/Fonts`), you can't make it relocatable (unless you put it into separate packages).

  For more information, see "Specify the Installation Destination" (page 47).
- How much will you want to customize the installation process? You can supply a welcome statement, a Read Me file, a license agreement, and a background image for the installer window. These are described in "Customizing What Installer Shows to the User" (page 61), which also describes how you can localize the information you provide for customization.

- Do you need to check for certain requirements on the installation system or the installation volume? Do you need other special processing before or after installing your software? See "Specifying Installation Requirements" (page 111) and "Customizing an Installation With Executable Scripts" (page 37) for information on how to perform these kinds of operations.
- Will installing your software require root or administrator authorization? For more information, see "Authorization, File Ownership, and Permissions" (page 79).
- Does your software require or recommend a restart, logout, or shutdown? You can specify such requirements in PackageMaker.

  See "Restarting, Logging Out, or Shutting Down" (page 39).

There are additional issues you may need to resolve for some software installations. These include:

- Will you provide information that can help Installer find your previously installed software during future upgrades? This is most likely to be an issue if your software is easily relocatable. For more information, see "Finding Previously Installed Software" (page 97).
- Will you provide information in your software that can help Installer decide what needs to be replaced during future upgrades? For more information, see "Specifying Version Information for Packaged Software" (page 107).

Finally, you should take a tour of the PackageMaker application, using the Help and help tags to gain familiarity with the interface and available options.

Roadmap

Once you have completed development of your software, including making any changes required by the issues described in "Preparation" (page 43), you're ready to start packaging. That will require the following steps:

1. Create a directory to store your software
2. Create a directory to store additional installation files
3. Create any files you need to customize the installation
4. Create files you need for special processing
5. Specify the installation destination
6. Create a package with PackageMaker
7. Modify the package for special processing
8. Test the package

These steps are described in the sections that follow.

Once you have packaged your software, you can distribute it with the mechanisms described in "Software Distribution Guidelines" (page 17).

Create a Directory to Store Your Software

You'll have to create a directory to store your software prior to packaging. This directory can have any reasonable name, but for this example, name it `Package_contents`, indicating it contains the software to be packaged. These files will end up in an archive file in the package. Use the Root field on the Files pane to tell PackageMaker about this directory.

If your software consists of one, atomic component, such as an application or framework bundle, you can place it directly in the `Package_contents` directory. Similarly, if your software consists of multiple components, but they are all installed in the same location, you can place each of the components directly in the `Package_contents` directory.

If your software consists of multiple components that must be placed in specific places in a directory hierarchy, you should create a directory structure within `Package_contents` that mirrors the installation hierarchy. For example, to install an application in `/Applications` and some fonts in `/System/Library/Fonts`, you set up a directory structure like the one shown in Listing 1.

Listing 1 Directory structure for installing software in two fixed places

```
Package_contents
    Applications
    System
        Library
            Fonts
```

You then place your application in the `Applications` directory and your fonts in the `Fonts` directory. Later in this process, you'll specify a location for installing your software, as described in "Specify the Installation Destination" (page 47). For this example (with an application and some fonts), you specify an installation destination of `/` (the root directory), and the files are installed in the appropriate subdirectories. For an installation that places all the files in one location, you can specify the exact location, such as `/Applications`.

If your software consists of multiple components that don't all have to be installed, you should put each component into a separate package, then combine the packages in a metapackage. For more information, see "Anatomy of a Metapackage" (page 75).

Create Files to Customize the Installation

You'll need to create any text and image files desired to customize the installation. Depending on your requirements, as described in "Preparation" (page 43), you may need some or all of the following:

- a welcome statement
- a Read Me file
- a software license agreement
- a background image for the installer window

For information on how to create and use these files, see "Customizing What Installer Shows to the User" (page 61).

Note: You may also want to provide information that can help Installer find your previously installed software during future upgrades. If so, you'll have to create a `TokenDefinitions.plist` file. For more information, see "Finding Previously Installed Software" (page 97)

Create Script Files for Special Processing

You'll need to create any script files (or other executables) you'll use to perform special handling during the installation.

Important: You may be able to minimize your use of scripts, which present potential security problems, by using techniques described in "Examining and Modifying Property Lists" (page 87) and "Finding Previously Installed Software" (page 97).

Depending on your requirements, as described in "Preparation" (page 43), you may need some or all of the following:

- an InstallationCheck file to check for required features of the installation system

  For details, see "Checking the Installation With InstallationCheck" (page 125).
- a VolumeCheck file to examine each mounted volume and determine whether it meets your required criteria

  For details, see "Checking the Installation Volume With VolumeCheck" (page 133).
- `preflight`, `postflight`, `preinstall`, `postinstall`, `preupgrade`, or `postupgrade` scripts to perform additional operations

  For details, see "Modifying an Installation With Scripts" (page 139).

Create a Directory to Store Additional Installation Files

You'll have to create a directory to store the files you've created for customizing the installation process and performing special processing. This directory can have any reasonable name, but for this example, name it `Install_resources`, indicating it contains additional resources to be used by Installer during the installation process. Use the Resources field on the Resources pane to tell PackageMaker about this directory.

Once you have created the `Install_resources` directory, you should copy in the files you created in "Create Files to Customize the Installation" (page 45) and "Create Script Files for Special Processing" (page 46).

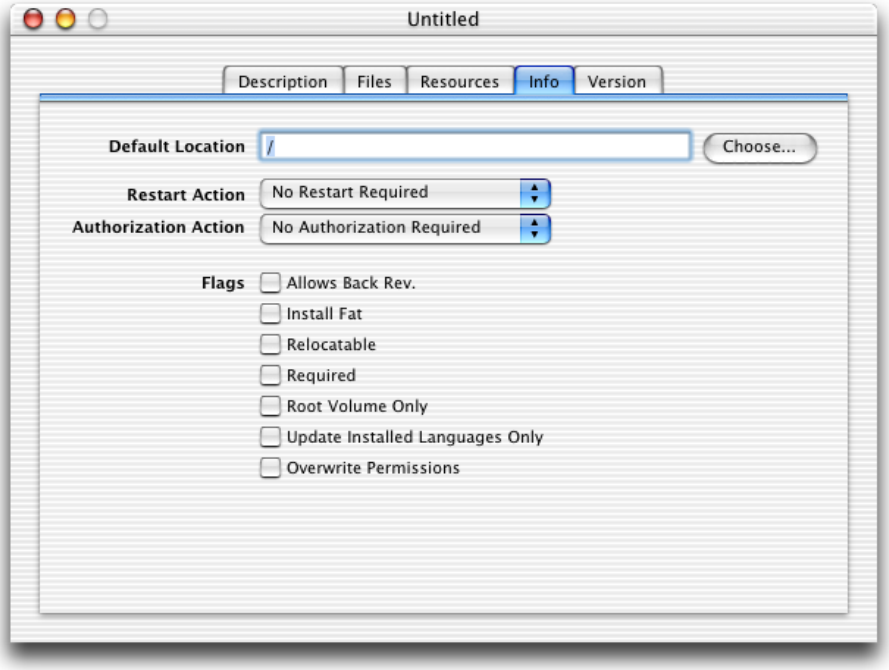
In some cases, you may want to create localized directories within the `Install_resources` directory. For details, see "Localizing the Welcome, Read Me, and License Files" (page 63), "Additional Tips for Background Images" (page 68), and "A Complex Installation Package" (page 71).

Specify the Installation Destination

You'll have to supply PackageMaker with a default destination for installing the software on the volume selected by the user. For example, you might specify a default location of `/Applications`. You supply this information in the Default Location field on the PackageMaker Info pane. Figure Figure 1 shows the Info pane with its default settings.

Note: For additional information on features listed on this pane, see "Restarting, Logging Out, or Shutting Down" (page 39) and "Authorization, File Ownership, and Permissions" (page 79), as well as the PackageMaker Help and help tags in the application.

Figure 1 The PackageMaker Info pane, with default settings



The following sections describe how to specify a fixed location, how to let the user specify the location, and how to specify the location for a package that will be part of a metapackage.

Installing in a Fixed Location

To require that your software be installed in a fixed location, observe the following:

- If your package places items in more than one location (such as an application in `/Applications` and fonts in `/System/Library/Fonts`, you should set the default location to the highest enclosing directory (in this case, `/`). If your package places items in only one location, you can choose any default location.
- Do not select the Relocatable checkbox shown in Figure 1.

- Place your software in a directory hierarchy within the `Package_contents` directory that is consistent with the default directory into which the software will be installed, such as in the example shown in Listing 1.

If you follow these recommendations, Installer will place your software only in the directory hierarchy you have specified and will not allow a user to choose a destination for the software. Installer assembles paths based on the mountpoint of the selected volume, plus the value of the default location, plus the paths in the structure in your `Package_contents` directory.

Important: If the directory structure in your `Package_contents` directory is not consistent with the default directory in which the software will be installed and the package is not relocatable, the installation will not succeed.

Letting the User Choose the Location

To allow a user to choose where to install your software, there are two requirements:

- Select the Relocatable checkbox shown in Figure 1.
- Place all your software directly within the `Package_contents` directory.

Important: To allow a user to choose where to install your software, you cannot put software into multiple locations, although you can specify a directory hierarchy in the package that can be installed, in its entirety, into an existing directory.

If you follow these two requirements, Installer will allow the user to choose where to install your software. Installer assembles paths based on the mountpoint of the selected volume, plus the location selected by the user, plus the paths in the structure in your `Package_contents` directory.

Note, however, that the user's choice of location may be constrained in several other ways:

- You can require that the installation be on the root volume (the one with the running system) with the Root Volume Only checkbox shown in Figure 1.
- You can require that the user have root or administrator privileges with the Authorization Action pop-up menu shown in Figure 1. If the user does not have the required privileges and cannot authenticate to receive authorization, the installation is canceled.
- Your package can include an InstallationCheck script that checks for various system requirements before allowing an installation to proceed.
- Your package may include a VolumeCheck script that checks each volume and disallows installation on any volume that doesn't meet your criteria.

Choosing the Location for a Metapackage

When you combine individual packages in a metapackage, the user will be able to choose a destination if any of the individual packages is relocatable. If this is the case, the user will be able to choose only one destination for all relocatable packages they install, not separate destinations for each relocatable package; installed packages that are not relocatable will not be relocated. Note, however, that prior to Mac OS X version 10.3.4, relocation did not work reliably for metapackages.

Create a Package With PackageMaker

When you launch PackageMaker, it automatically opens an untitled package editor window. You can also choose File > New > Single Package to open a new package editor window.

When you have filled in all the desired information, choose File > Create Package (or Command-K) to create an installation package for your software. For convenience, you may want to save the package in the same directory where you created your `Package_contents` and `Install_resources` directories.

If a package is lacking required information (for example, you forgot to specify the location of the software files to include in the package), PackageMaker displays a warning and will not create the package. On success, PackageMaker displays a message that the package was assembled without any problems. However, there may still be problems with the package—see "Test the Package" (page 49) for more information.

You should also save the package definition file for your package. You can give it the same name as your package file and store it in the same directory. For example, you might have `MyGreatSoftware.pkg` and `MyGreatSoftware.pmsp`. Double-clicking the latter opens a package editor window for the former, where you can make changes and create an updated package.

Modify the Package for Special Processing

There are currently some installation features that are not available through the PackageMaker user interface. To take advantage of these features, you will have to directly modify files within the package. For information on these features, and on how to modify files in the package, see:

"Supplying a Background Image" (page 64)
"Examining and Modifying Property Lists" (page 87)
"Other Keys You Might Want to Modify" (page 96)
"Finding Previously Installed Software" (page 97)
"Specifying Installation Requirements" (page 111)

Test the Package

When you have successfully created an installation package, you'll naturally want to test it by attempting to install the software. If PackageMaker did not report any errors, the package should be valid, but there may still be problems, or modifications you'll want to make to the package.

Double-clicking a package launches Installer. If Installer detects an invalid package, it displays an error message and stops the installation. This is an early and useful aid in debugging a package.

Warning: When an installation is canceled during testing, Installer remains open with its current (invalid) state, so you should be sure to quit Installer before further testing. Otherwise, you can modify the package, test again, and see the same error because of the saved state.

In addition to the testing information in the following sections, see "Troubleshooting Packages and Installation" (page 145).

Viewing the Installer Log and Script Output

Another important testing aid is the Installer Log window, in which Installer displays a log of various operations it performs. Even if an installation appears to complete successfully, there may be information in the Installer Log window revealing possible errors (such as permissions problems). And even when Installer displays an error alert, information in the log window may elaborate on the alert.

To open the log window, choose File > Show Log. The Installer Log window has buttons for printing and saving its information. In addition, the log window displays output from the installation script files described in "Modifying an Installation With Scripts" (page 139).

Output for the InstallationCheck and VolumeCheck scripts (described in "Checking the Installation With InstallationCheck" (page 125)"Checking the Installation Volume With VolumeCheck" (page 133)) is directed to the Console window. The Console application is located in `/Applications/Utilities`.

Many of the previously mentioned scripts can halt execution by returning a non-zero value. When this happens, Installer will display an error message. However, you should again look for additional information in the appropriate places (the Installer Log window or Console window). In addition, if your scripts write progress or error information to a log file, you should check that file too.

Common Problems Found in Testing

Some of the common problems you may find when testing your package include the following:

- The software wasn't installed where you expected.

  See "Specify the Installation Destination" (page 47)

- Installation halts with the message that a script or tool is not set executable and readable.

  If you assemble a package with PackageMaker, it will automatically set correctly named scripts to be executable and readable. If you modify a package file directly, it's your responsibility to ensure that the script file is both executable and readable.

- Installation halts due to a permissions problem.

  You may have specified an installation location for which the user doesn't have sufficient permissions privileges, but you haven't required Installer to request authorization. For more information, see "Authorization, File Ownership, and Permissions" (page 79)

- A requested action didn't occur, such as a request for authorization, or a restart on completion of the installation.

Open the package definition file (`.pmsp`) and verify that you did in fact specify the missing action. You can also examine the Installer Log and any script logs for clues as to what did actually happen.

The Installation Process

This article describes factors that can affect an installation. It also lists the typical steps that take place in the installation process.

The Installer application resides in `/Applications/Utilities` and is included with Mac OS X. Double-clicking an installation package file or a metapackage file launches Installer, which then takes the user through the installation process. That process can vary substantially, depending on the information the developer has supplied in the package file and the actions taken by the user.

Factors That Affect an Installation

The actual operations performed for a given installation depend on

- whether the installation is for a package or a metapackage
- whether the package or metapackage contains files such as the following for customizing the information Installer shows to the user:
  - `Welcome.rtf`
  - `Readme.rtfd`
  - `License.txt`

  The presence of these files (in the correct location, and with the correct spelling, including case and extension) causes Installer to present the information they contain to the user. If the files are not present or their names are misspelled, Installer shows only its default information. For information on how to use these files, see "Customizing What Installer Shows to the User" (page 61).
- whether the package's information property list contains requirements information, which must be checked before installing (an option available starting with Mac OS X version 10.3)

  This feature is described in "Specifying Installation Requirements" (page 111).
- whether executable files with the following names are present in the package's `Resources` directory:
  - `InstallationCheck` (described in "Checking the Installation With InstallationCheck" (page 125))
  - `VolumeCheck` (described in "Checking the Installation Volume With VolumeCheck" (page 133))
  - `preflight`, `preinstall`, `preupgrade`, `postinstall`, `postupgrade`, `postflight` (described in "Modifying an Installation With Scripts" (page 139))

  These executables are optional. If they are present, in the right location and with the names spelled correctly, including case, they are executed. If an executable is present but its name is misspelled, it is ignored and no error is reported.

  These files must be both executable and readable.

- whether the installation is an upgrade (a receipt file for the package exists in the `/Library/Receipts` directory of the destination volume, meaning the software has been installed previously) or an install (no receipt exists)
- whether the installation package is relocatable
- various interactions between the user and Installer, such as
  - authenticating (if required)
  - accepting a license agreement (if required)
  - selecting a volume
  - selecting a destination directory (if allowed)
- whether the installation package is valid:
  - a package created without problems in PackageMaker is generally valid, though that is not guaranteed
  - a package may be invalid for various other reasons, such as incorrect manipulation of its contents or incompatibilities with the user's system

  If Installer detects an invalid package, it displays an error message and stops the installation. If this happens during package testing, you should quit Installer before further testing.
- whether the installation contains any resource forks Installer must reassemble, as described in "Preserving Resource Fork Data" (page 40)

"Installation Steps, in Order" (page 54) shows the actual steps for an installation, subject to the factors listed above.

Installation Steps, in Order

The following list shows the approximate order of operations for an installation. Steps marked with the • symbol are for metapackages only. "Factors That Affect an Installation" (page 53) describes factors that can affect whether certain steps are performed or not.

Important: This information is provided as a help in understanding the installation process. It is not guaranteed to be correct in all cases, and is subject to change.

1. A user double-clicks the icon for an installation package file.
2. Installer is launched and opens the package.
3. An Authorization pane is displayed and the user is asked to authenticate, if required.

   Failure to authenticate cancels the installation.
4. InstallationCheck is run, if present.

   • For a metapackage, InstallationCheck is run for every package that contains the script. If the metapackage itself contains the script, it is not run.

If InstallationCheck returns anything but zero, the installation is canceled.

> **Note:** Starting in Mac OS X version 10.3 (v10.3), if the information property list for a package contains any requirements specifications, Installer ignores InstallationCheck (if present) and makes a pass through all the requirements, ignoring any requirements that are volume-specific. For more information, see "New and Existing Requirements Checking" (page 111).

5. The Introduction (or Welcome) pane is displayed. If the file `Welcome.rtf` is present, its text is displayed; otherwise, Installer uses default text.

   If the file `background.tiff` (or any of several other supported image types) is found, it is used as a background image in the installer window.

6. The Important Information (or Read Me) pane is displayed only if the file `ReadMe.rtf` is present.

7. The Software License Agreement pane is displayed only if the file `License.rtf` is present.

   If the user does not agree to the license, the installation is canceled.

   • For a metapackage, a License pane is displayed only if the metapackage itself contains a license file. License agreements in contained packages are not shown.

8. On entering the Select a Destination pane:

   For a package: the VolumeCheck script is run, if present, for each mounted volume.

   • For a metapackage:

   a. The VolumeCheck script is run, if present, for each mounted volume.

   b. The VolumeCheck script for each contained package is run, if present, for each mounted volume.

   If any script returns a value other than 0 for any volume, the icon for that volume is badged to indicate it cannot be used for installation. If a user clicks that icon, Installer displays an error message.

   > **Note:** Starting in Mac OS X v10.3, if the information property list for a package contains any requirements specifications, Installer ignores VolumeCheck (if present) and makes a pass through all the requirements, once for each volume, ignoring any requirements that are *not* volume-specific. For more information, see "New and Existing Requirements Checking" (page 111).

9. The user selects a destination volume for the installation.

   If the software is relocatable, the user can also pick a destination directory.

   • For a metapackage, the user can pick a destination directory if the software in any contained package is relocatable.

10. Installer determines whether to perform an install or an upgrade.

11. • For a metapackage only, the user can click the Customize button.

    Installer displays the Custom Install pane, where the user chooses which items to install or upgrade.

The user can click a button to go back and forth between the Custom Install and the Easy Install.

12. The user clicks the Install/Upgrade button.

> **Note:** This list of steps does not show the order in which the scripts that follow are run when installing a metapackage. For that information, see "Execution Order for a Complex Metapackage" (page 56).

13. If `preflight` is present, it is run.

    If `preflight` returns anything but 0, the installation is canceled.

14. Depending on whether the installation is an install or an upgrade, `preinstall` or `preupgrade` is run, if present.

    If `preinstall` or `preupgrade` returns anything but 0, the installation is canceled.

15. The package contents are extracted and copied to the destination volume.

16. An installation receipt is copied to the receipts directory (`/Library/Receipts`) on the destination volume. The receipt lists the files that were installed, but not the destination (for relocatable packages).

17. `postinstall` or `postupgrade` is run, if present.

    If `postinstall` or `postupgrade` returns anything but 0, the installation is canceled.

18. `postflight` is run, if present.

19. Installer handles restart, logout, or shutdown, if requested by the package.

    • For a metapackage, if any package selected for installation requires a restart action, Installer handles it.

> **Note:** Recall that no restart action takes place unless the user chooses to install a package on the currently booted volume.

Execution Order for a Complex Metapackage

This section shows the order of execution of the various pre-and postprocessing script files during installation of a complex metapackage, shown in Figure 1 (page 30). Listing 1 shows the contents of the Umbrella metapackage, which lists one metapackage (Two_Apps) and one package (WebObjects Documentation). TwoApps metapackage in turn lists two applications (Cool_App and Mouse_Pad).

Listing 1 The contents of the Umbrella metapackage

```
TwoApps metapackage
    Cool_App
    Mouse_Pad
WebObjects Documentation
```

When each of the packages and metapackages shown in Listing 1 contained the full complement of pre-and post-processing script files (`preflight`, `preinstall`, `preupgrade`, `postinstall`, `postupgrade`, and `postflight`), the order of execution during an install (the software was installed for the first time) was as shown in the following list:

Install

1. Umbrella metapackage preflight
2. Two_Apps metapackage preflight
3. Cool_App preflight
4. Mouse_Pad preflight
5. WebObjects preflight
6. Umbrella metapackage preinstall
7. Two_Apps metapackage preinstall
8. Cool_App preinstall
9. Cool_App postinstall
10. Mouse_Pad preinstall
11. Mouse_Pad postinstall
12. Two_Apps metapackage postinstall
13. WebObjects preinstall
14. WebObjects postinstall
15. Umbrella metapackage postinstall
16. Umbrella metapackage postflight
17. Two_Apps metapackage postflight
18. Cool_App postflight
19. Mouse_Pad postflight
20. WebObjects postflight

When the metapackage shown in Listing 1 was run again, the order of execution for an upgrade (the software had been previously installed) was as shown in the following list:

1. Umbrella metapackage preflight
2. Two_Apps metapackage preflight
3. Cool_App preflight
4. Mouse_Pad preflight

5. WebObjects preflight
6. Umbrella metapackage preupgrade
7. Two_Apps metapackage preupgrade
8. Cool_App preupgrade
9. Cool_App postupgrade
10. Mouse_Pad preupgrade
11. Mouse_Pad postupgrade
12. Two_Apps metapackage postupgrade
13. WebObjects preupgrade
14. WebObjects postupgrade
15. Umbrella metapackage postupgrade
16. Umbrella metapackage postflight
17. Two_Apps metapackage postflight
18. Cool_App postflight
19. Mouse_Pad postflight
20. WebObjects postflight

To summarize from these results:

- If a metapackage contains multiple packages, all of the scripts are run in the order in which they are listed in the metapackage's package list.
- Metapackage scripts such as `preflight` and `postupgrade` (even for embedded metapackages) are run once for the entire set of packages.

This example does not include execution order for the InstallationCheck and VolumeCheck scripts. That order is described in "Installation Steps, in Order" (page 54), and can be summarized as follows:

- Both of these scripts are run, if present, before any of the pre- and post scripts.
- For a package, the InstallationCheck script is run, if present.

  For a metapackage, for every contained package that has an InstallationCheck script, the script is run. If the metapackage itself has an InstallationCheck script, that script is not run.
- For a package, the VolumeCheck script is run, if present, for each mounted volume.

  For a metapackage

  1. The VolumeCheck script for the metapackage is run, if present, for each mounted volume.
  2. The VolumeCheck script for each contained package is run, if present, for each mounted volume.

3. If a volume fails testing by the VolumeCheck script from any package or metapackage, it is presented as not valid for the installation package.

Note: Starting in Mac OS X version 10.3, requirements checking may take the place of executing the InstallationCheck and VolumeCheck scripts. For information on the order in which requirements checking takes place, see "New and Existing Requirements Checking" (page 111).

Customizing What Installer Shows to the User

You can customize what the Installer application shows to the user by the information and resources you build into your installation package with PackageMaker. You can optionally supply localizable files that provide text for the following Installer panes:

- Introduction (or Welcome)
- Read Me
- License

In addition, you can supply a localizable background image for the installer window and control how it is displayed.

Finally, you must provide a title and should also provide a description for your installation package. You can provide similar information for a metapackage.

For information on the order in which various Installer panes are displayed, see "Installation Steps, in Order" (page 54).

Note: The information in the sections that follow applies to both packages and metapackages, unless stated otherwise.

Supplying a Title and Description

For a package or a metapackage, you supply a title and description in, respectively, the Title and Description fields of the PackageMaker Description pane.

The title is required and is displayed both in the installer window title and above the Introduction pane, in a sentence of the form "Welcome to the <title> Installer". For example, Figure 1 (page 62) shows the result of supplying "Simple Carbon Application" as a package title. This figure also shows the default Welcome text you see if you do not supply your own text. If you supply a title that is too long, it currently is clipped in the window title and truncated in the Introduction pane.

Figure 1 The Introduction pane for the Simple Carbon Application

The information you supply in the Description field for a package does not appear in any Installer pane if the package is installed individually. Rather, it appears when the package is part of a metapackage installation, and the user selects the package in the Custom Install pane.

Similarly, the information you supply in the Description field for a metapackage appears only if the metapackage is part of a metapackage installation and the user selects the metapackage in the Custom Install pane, as shown in Figure 1 (page 30).

Supplying Text for the Introduction, Read Me, and License Panes

Installer always shows an Introduction (or Welcome) pane, followed by Read Me and License panes if the package or metapackage contains files for those panes. For a complete picture of the order in which panes are displayed, see "Installation Steps, in Order" (page 54).

You can supply text for these panes by providing localizable files named, respectively, Welcome, Read Me, and License, with the appropriate extension. The supported extensions (and file formats) are `.txt` (text file), `.rtf` (Rich Text Format), `.rtfd` (Rich Text Format Directory), and `.html` (Hypertext Markup Language). For example, a ReadMe file might be named `ReadMe.rtf`. Case is significant so, for example, Installer will not display a Read Me named `Readme.rtf`.

If you supply more than one of these files with the same name but different extensions, Installer searches by extension in this order: `.rtfd`, `.rtf`, `.html` and `.txt`. So if you supply two license files named `License.rtf` and `License.rtfd`, Installer will display the contents of `License.rtfd` on the License pane.

The welcome, Read Me, and license files reside in `Resources` directory of a package, or in a localized directory within the `Resources` directory. Before creating a package, you should place these files, along with any other required resources, in the `Install_resources` directory, as described (for packages) in "Creating a Package" (page 43).

Listing 1 shows an `Install_resources` directory on your hard disk (before creating a package) that includes welcome, Read Me and license files that are not localized. For information on localizing the customized information you supply, see "Localizing the Welcome, Read Me, and License Files" (page 63).

Listing 1 A resource directory with files to customize an installation

```
- Install_Resources
        License.rtf
        ReadMe.rtfd
        Welcome.rtfd
```

When packages are included in a metapackage, only files associated with the metapackage are shown. For example, the license file in an individual package is not shown—only a license file associated with the metapackage is shown.

The Installer Introduction pane does not scroll, so the welcome file should provide a relatively brief description of the package or metapackage. A user should be able to quickly determine whether to install the software. If you do not provide a welcome file, Installer displays a default Introduction pane.

The Read Me pane is scrollable and can be printed. The Read Me file can be lengthy. For example, it may provide detailed information about hardware or software requirements, new product features, or other issues. For a metapackage that includes dissimilar types of software, the Read Me file for the metapackage should make clear why the packaged software is distributed together. If you do not provide a Read Me file, Installer does not display a Read Me pane.

The license file provides your license agreement. If the file is present, the user will be required to accept the agreement before continuing with the installation. If you do not provide a license file, Installer does not show a license agreement.

Localizing the Welcome, Read Me, and License Files

If you have only one version of your welcome, Read Me, and license files, you should put them directly inside the `Install_resources` directory, as described in "Creating a Package" (page 43), before creating your package (or metapackage). However, if you want to localize any of these files, you can put copies in the desired localized directories within your main `Resources` directory. As a result, Installer displays the appropriate file according to the user's current Languages preference.

Files you place in your main `Resources` directory end up in the main `Resources` directory of the package. Files you place in a localized subdirectory end up in a similarly named localized subdirectory in the package.

If you place a default welcome file in a package's main `Resources` directory, Installer uses that file if it can't find an appropriate localized welcome file for the user's current language setting. For the Read Me and license files, however, if the package has any versions in localized directories, Installer shows only one of those, and does not show a version in the main `Resources` directory.

Listing 2 shows an `Install_resources` directory on your hard disk that includes Read Me and license files localized for English and French (in the `English.lproj` and `French.lproj` subdirectories, respectively). It also includes a welcome file at the highest level (not within a localized directory). The welcome file is used

in all cases, and the Read Me and license files are chosen as appropriate for the user's language setting. For additional information on localizing the customized information you supply, see "Additional Tips for Background Images" (page 68).

Note: If a package supplies multiple localized license files, the License pane offers the user a pop-up menu of language choices.

Listing 2 A resource directory with files to customize an installation

```
- Install_Resources
    - English.lproj
        License.rtf
        ReadMe.rtfd
    - French.lproj
        License.rtf
        ReadMe.rtfd
      Welcome.rtfd
```

Supplying a Background Image

Starting with the Installer version distributed with Mac OS X version 10.2.0, packages and metapackages can contain a custom, localizable background image that is displayed in the installer window during installation. If a package or metapackage does not contain a background image, Installer displays the default image of a CD with a large "X".

You can specify both an alignment setting and a scaling setting for a background image. However, these settings are not yet supported in PackageMaker. For all these reasons, you must first create your package, then set the appropriate keys by editing the package's `Info.plist` file. For more information on working with such keys and on how to edit the `Info.plist` file, see "Examining and Modifying Property Lists" (page 87).

Naming the Background Image

The background image file must be named `background`, with one of the following extensions: `.jpg`, `.tif`, `.tiff`, `.gif`, `.pict`, `.eps`, and `.pdf`. For example, a background image might have the name `background.jpg`. Case is significant. For example, Installer will not display an image named `background.JPG`.

Package Placement for the Background Image

The background image resides in a package's `Resources` directory, as shown in Listing 3, or in a localized directory within the `Resources` directory. Before creating a package, you should place the background image file, along with any other resources for the package, in the `Install_resources` directory, as described in "Creating a Package" (page 43). For information on localizing, see "Additional Tips for Background Images" (page 68).

Listing 3 Package contents with nonlocalized background image

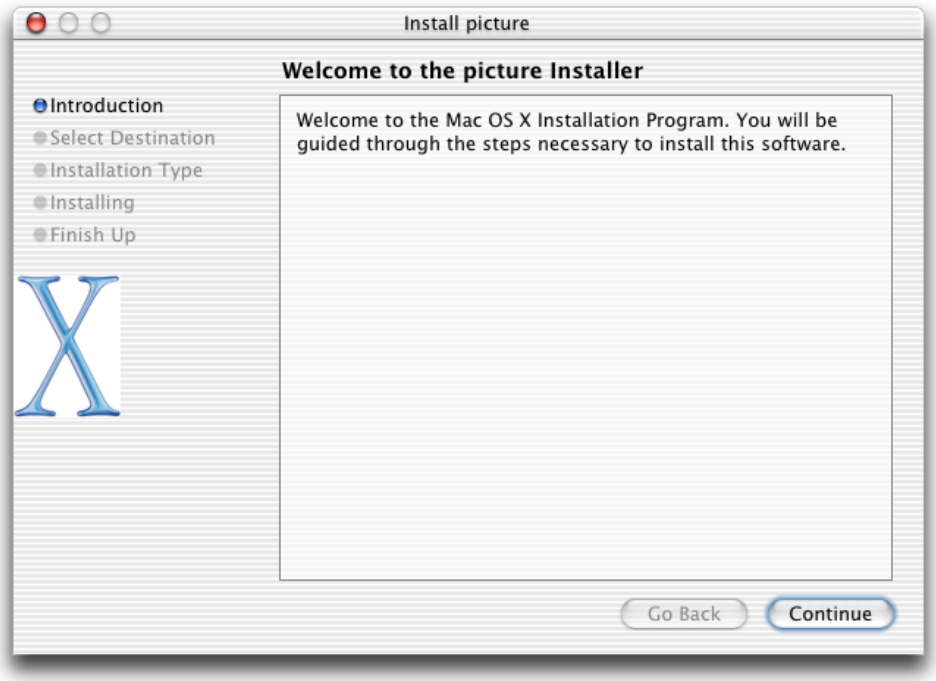
```
Contents
    Archive.bom
    Archive.pax.gz
    Info.plist
    Resources
        Description.plist
        background.tif
```

Aligning the Background Image

The key for aligning a background image is `IFPkgFlagBackgroundAlignment`, and it takes any of the following string values: `topleft`, `top`, `topright`, `left`, `center`, `right`, `bottomleft`, `bottom`, `bottomright`. These values align the image as their names suggest.

Figure 2 shows an installer window with an image of the Mac OS X logo, positioned with `left` alignment.

Figure 2 A background image aligned left



Listing 4 shows the alignment key from the XML listing of an `Info.plist` file for a package.

Listing 4 The IFPkgFlagBackgroundAlignment key from an Info.plist file

```
<key>IFPkgFlagBackgroundAlignment</key>
<string>bottomright</string>
```

You typically use alignment when you have a background image, such as an icon, that is significantly smaller than the installer window (as in the example in Figure 2). Alignment allows you to place it in a location that is appropriate for your software.

Scaling the Background Image

The scaling key controls how the background image fits within the installer window. The image can be used at its natural size, be scaled to fit the window exactly, or be scaled proportionally to fit the window as best it can without losing its natural proportions. The key for scaling a background image is `IFPkgFlagBackgroundScaling`, and it takes any of the following string values:

`none`
: The size and proportions of the image don't change. If the frame is too small to display the whole image, the edges of the image are trimmed off.

`tofit`
: The image shrinks or expands, and its proportions distort, until it exactly fits the frame.

`proportional`
: If the image is too large, it shrinks to fit inside the frame. If the image is too small, it expands. The proportions of the image are preserved.

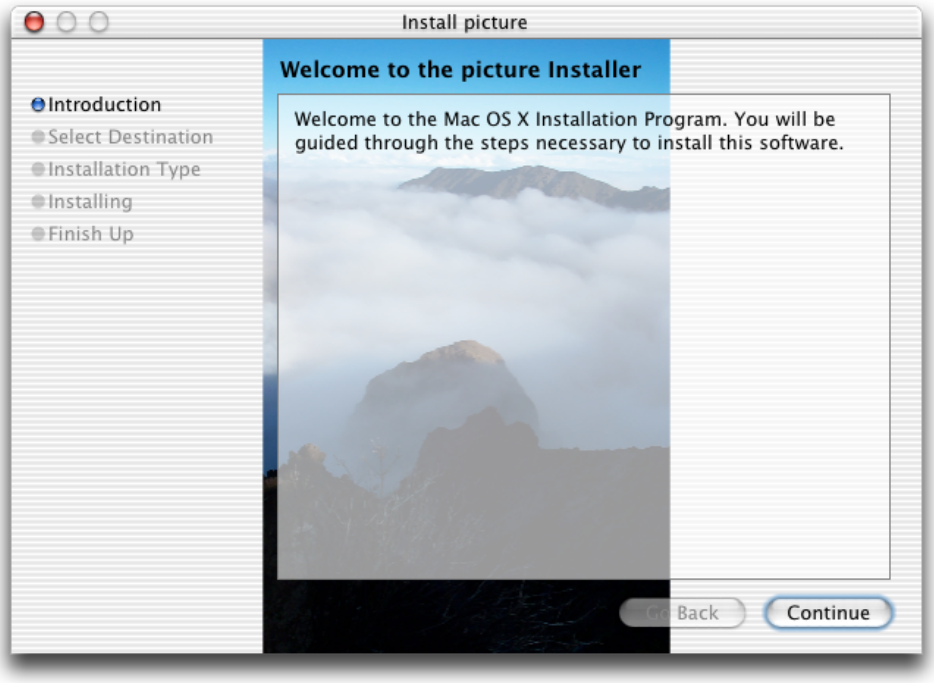
Listing 5 shows the background scaling key from the XML listing of an `Info.plist` file for a package.

Listing 5 The IfPkgFlagBackgroundScaling key from an Info.plist file

```
<key>IfPkgFlagBackgroundScaling</key>
<string>proportional</string>
```

Figure 3 shows the installer window with a background image scaled with an `IFPkgFlagBackgroundScaling` key value of `proportional`. It preserves the proportions of the original photo, and scales the photo until it fits inside the installer window without distortion or stretching. The original image is larger than the installer window, and you can see that it is taller than it is wide, so the result of scaling proportionally is that the image fills only a "column" in the middle of the window.

Figure 3 A background image scaled proportionally



Using a value of `tofit` would stretch (and distort) the photo to fill the window, while using `none` displays the image at full size so that only a small portion of it would be visible (remember, the photo is larger than the installer window).

Localizing the Background Image

If your package requires localized graphic content, you can place localized versions of the background image in multiple localized directories within a package's `Resources` directory.

Important: Localizing a large image can dramatically increase the size of your installation package. If you're distributing software on a CD, that may be no problem. However, if you're distributing software through Internet downloads, you may want to use smaller images and more compact image types.

Listing 6 shows the contents of a package that provides a localized background image for English, French, and Japanese, as well as a default image at the highest level in the `Resources` directory.

Listing 6 Package contents with a localized background image

```
Contents
    Archive.bom
    Archive.pax.gz
    Info.plist
    Resources
        background.tif
        English.lproj
            Description.plist
```

```
        background.tif
    French.lproj
        Description.plist
        background.tif
    Japanese.lproj
        Description.plist
        background.tif
```

Additional Tips for Background Images

The following tips may help you to achieve a more pleasing result with your background image:

- Use a picture that is just large enough to contain the background element. Do not use an image as large as the installer window if the image will not occupy the entire background of the window.
- Use the alignment key (described in "Aligning the Background Image" (page 45)) to position the picture properly. Doing so increases the likelihood that the image will be presented reasonably if the installer window becomes resizable in a future release.
- To have your window background appear with the standard Aqua background, use transparency when drawing. Do not attempt to use a background image for this purpose.
- The background of the content area is lighter than the rest of the background. This effect is created by Installer, so you don't need to make any special provisions for the area in your background image where the content is displayed. Again, if the installer window becomes resizable or the content area changes size or shape, you do not want your background image to become "inconsistent" because it assumed a specific size for foreground elements.

Anatomy of a Package

This section provides a detailed description of the contents of a complex installation package. It assumes you are familiar with the information in "About Packages" (page 29), which introduces packages and describes the contents of a simple package.

Important: This section describes the format for a package created with PackageMaker version 1.1.10 or later. Version 1.1.10 first shipped with the December 2002 Developer Tools CD.

An installation package is actually a directory, with extension `.pkg`, that is presented to the user as a file. You create a package with PackageMaker. The information you supply through its user interface is stored in the package. This process is described in detail in "Creating a Package" (page 43).

Double-clicking a package launches Installer. The package contains all the information Installer needs to take a user through the process of installing the packaged software.

A package has a well-defined format, with required and optional subdirectories and files. To display the contents of an installation package (or any other directory displayed in the Finder as a single file), Control-click the file and choose Show Package Contents from the contextual menu.

Note: You can display the contents of a package in a Terminal window with the `find` command. For an example, see Listing 2 (page 88).

High-Level Package Structure

Every installation package contains one directory at the highest level, named `Contents`. Within that directory, all packages contain several files and a `Resources` directory, as shown in Listing 1. The filenames are always the same, except that if the archive file is not compressed, it does not have the `.gz` extension.

The files at the highest level in the package contain the software to install and information about where to install it (`Archive.bom` and `Archive.pax.gz`), as well as information about the installation (`Info.plist`). The `PkgInfo` file provides backward compatibility with previous versions of Installer.

Listing 1 Files and directories in a package's Contents directory

```
Archive.bom
Archive.pax.gz
Info.plist
PkgInfo
Resources
```

The structure of the `Resources` directory is also fairly straightforward. It contains a relatively small number of files, and may also contain one or more localized directories. A `Resources` directory can contain these kinds of files:

- A `Description.plist` property list file, which provides additional information about the installation.
- Files for customizing the Installer user interface, such as a Read Me file or background image file.
- Executable files that provide additional control over the installation process, such as `InstallationCheck`, `VolumeCheck`, `preflight`, and `postflight` scripts.
- Files that provide backward compatibility with previous versions of Installer, such as a `.sizes` file and files that link to other files in the `Contents` directory.

A `Resources` directory's localized directories, such as `French.lproj` and `English.lproj`, can contain the following:

- Localized versions of the `Description.plist` property list file
- Localized versions of the same customizing files in the `Resources` directory itself, such as a Read Me or license file
- Localized message files that contain text shown to the user as a result of executing an `InstallationCheck` or `VolumeCheck` script: `InstallationCheck.strings` and `VolumeCheck.strings`

Less commonly, a package can contain the files described in "Additional Package Files" (page 73).

All of these files are described in more detail in the following sections.

Localized Folder Names

You can use any of the folder names shown in Table 1 to create localized folders in the `Resources` folder in a package. If you construct other folder names from the ISO 639 standard for language codes and the ISO 3166 standard for country codes, they may not be recognized by Installer when installing your package.

Table 1

Localized folder names
`Dutch.lproj`
`English.lproj`
`French.lproj`
`German.lproj`
`Italian.lproj`
`Japanese.lproj`
`Spanish.lproj`
`da.lproj`
`fi.lproj`

Localized folder names
ko.lproj
no.lproj
pt.lproj
sv.lproj
zh_CN.lproj
zn_TW.lproj

A Complex Installation Package

Figure 1 shows the contents of a complex package that contains examples of all the types of files a package typically contains, except those described in "Additional Package Files" (page 73). You can create a package like this one by following the steps in "Creating a Package" (page 43). These files and directories include:

- `Archive.bom`: A binary Bill of Materials file (hence the `.bom` extension) that describes the files in the `Archive.pax` or `Archive.pax.gz` file.

  For a receipt file (a kind of package described in "Installs and Upgrades" (page 33)), the `.bom` file lists the files that were actually installed. This can differ from the `.bom` file in the original package for two reasons:

  - If the package was relocatable, the paths to the installed files may be different from the paths in the original package.
  - If the installation is an upgrade, in certain cases Installer may not install all the files in the package. For more information, see "Specifying Version Information for Packaged Software" (page 107).

- `Archive.pax.gz`: A compressed archive file that contains the software to be installed. If the archive is not compressed, the `.gz` extension is omitted.
- `Info.plist`: An information property list file containing information from PackageMaker about the installation. The information is stored as a list of keys and associated values.

  For more information see "Examining and Modifying Property Lists" (page 87).
- `Pkginfo`: A file containing file type and creator information. Included for compatibility with earlier versions of Installer.

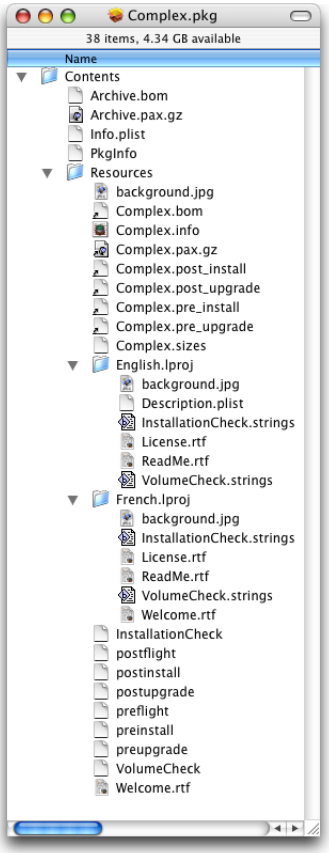
The package's `Resources` directory contains the following files:

- `background.jpg`: An image file to be used as the background in the installer window if the user's Languages preference does not match any of the localized directories.

  For information on how to provide and position a background image, see "Supplying a Background Image" (page 64).
- `complex.bom`: A link to the `Archive.bom` file. Included for compatibility with earlier versions of Installer.

- `complex.info`: Combines information found in the `Info.plist` and `Description.plist` files. Included for compatibility with earlier versions of Installer.
- `complex.pax.gz`: A link to the `Archive.pax.gz` file. Included for compatibility with earlier versions of Installer.
- `complex.post_install`, `complex.post_upgrade`, `complex.pre_install`, and `complex.pre_upgrade`: Included for compatibility with earlier versions of Installer. See `postinstall`, `postupgrade`, `preinstall`, and `preupgrade` for a description of the current versions of these files.
- `complex.sizes`: Contains information on the installed size of the package. Starting with the version of PackageMaker distributed with Mac OS X version 10.2.0, you can determine the size by querying the `Archive.bom` file. Included for compatibility with earlier versions of Installer.

Figure 1 The contents of a complex installation package



- `English.lproj`: Contains resources localized for English. In this case, these include:
  - `background.jpg`: Provides a background image that is displayed by the Installer on every pane.

    For more information, see "Supplying a Background Image" (page 64).
  - `InstallationCheck.strings` and `VolumeCheck.strings`: Provide localized message files that contain text shown to the user as a result of executing an `InstallationCheck` or `VolumeCheck` script.

For more information, see "InstallationCheck Message Strings" (page 129) and "VolumeCheck Message Strings" (page 136).

- ❏ `License.rtf` and `ReadMe.rtf`: Provide localized text for the Installer License pane and Read Me pane.

  For more information, see "Supplying Text for the Introduction, Read Me, and License Panes" (page 62).

  Note that `English.lproj` does not contain a welcome file and relies on the welcome file in the `Resources` directory.

- `French.lproj`: Contains resources localized for French. In this case, the localized French directory contains the same files as the English directory, with one exception:
- ❏ `Welcome.rtf`: Provides localized text for the Installer Introduction (or Welcome) pane.

  For more information, see "Supplying Text for the Introduction, Read Me, and License Panes" (page 62).

- `InstallationCheck`: An executable file, typically a shell script, that Installer runs early in the installation process. You can supply this file to check for criteria such as minimum hardware requirements.

  For more information, see "Checking the Installation With InstallationCheck" (page 125).

- `postflight`, `postinstall`, `postupgrade`, `preflight`, `preinstall`, `preupgrade`: Executable files, run by Installer at specific times, that provide additional control over the installation process.

  For more information, see "Modifying an Installation With Scripts" (page 139).

- `VolumeCheck`: An executable file, typically a shell script, that Installer runs once for each mounted volume. You can supply this file to check for volume criteria and restrict the volumes a user can install on.

  For more information, see "Checking the Installation Volume With VolumeCheck" (page 133).

- `Welcome.rtf`: Provides nonlocalized text for the Installer Introduction (or Welcome) pane. Whenever the user's Languages preference is set for a language for which the package doesn't provide a localized welcome file, Installer uses this version. If there is no welcome file in the `Resources` directory, Installer uses its own default text.

  For more information, see "Supplying Text for the Introduction, Read Me, and License Panes" (page 62).

Additional Package Files

In addition to the files described in "A Complex Installation Package" (page 71), a package may also contain the following files:

- `BundleVersion.plist`: A property list file created by PackageMaker when it creates an installation package. The file contains the contents of any version property list files found by scanning the items contained in the archive of files to be installed.

  This type of file is described in "Specifying Version Information for Packaged Software" (page 107).

- `TokenDefinitions.plist`: A property list file you create that specifies the search methods Installer should use to find files from a previous installation during an upgrade.

This type of file is described in "Tokens Definitions and Path Mappings" (page 97).

Anatomy of a Metapackage

This section provides a detailed description of the contents of an installation metapackage. It assumes you are familiar with the information in "What Is a Metapackage?" (page 29).

You use package files for installing software and metapackage files for grouping multiple packages, which allows a user to perform a custom installation. You can also use PackageMaker to create package and metapackage definition files (with extensions `.mpsp` and `.pmsm`, respectively) to store the information you've entered, and to modify and recreate packages and metapackages.

High-level Metapackage Structure

A metapackage file is similar in construction to a package file, but instead of containing the software to be installed, it contains information about the packages (or other metapackages) to install. Listing 1 shows the contents of a simple metapackage.

Listing 1 Contents of a simple metapackage

```
Contents
Info.plist
PkgInfo
Resources
    Description.plist
    License.rtf
    ReadMe.rtf
    Simple Meta.bom
    Simple Meta.info
    Welcome.rtf
```

You use the PackageMaker metapackage editor to create a metapackage. To open it, you choose File > New Meta Package. You then enter information, such as the name of the metapackage, the packages or metapackages it contains, and the location of its resource folder (needed only if you provide License, Read Me, or other customizing information, as described in "Customizing What Installer Shows to the User" (page 61)).

Of course, you'll create your packages first, before adding them to a metapackage. One of the main advantages of a metapackage is that it can allow a user to do a custom installation and choose from among various items to install. To give the user more control over the process, it is worthwhile to break your software into as many pieces as you reasonably can, rather than create a single, monolithic package. For example, you might place your application, documentation, fonts, and other items in separate packages.

A metapackage shows, in the Custom Install pane, which packages will be installed and which are upgrades. The button to start the installation is labeled Install only if all of the packages have not yet been installed; otherwise it is labeled Upgrade. Items in a metapackage are installed in the order listed.

Choosing an Installation Location for the Software in a Metapackage

When you combine individual packages in a metapackage, the user will be able to choose a destination if any of the individual packages is relocatable. If this is the case, the user will be able to choose only one destination for all relocatable packages they install, not separate destinations for each relocatable package; installed packages that are not relocatable will not be relocated. Note, however, that prior to Mac OS X version 10.3.4, relocation did not work reliably for metapackages.

When a Package or Metapackage Is Required

You can mark a package as required on the Info pane in PackageMaker. When you add a package to a metapackage in PackageMaker, you can also set the Attribute field for the package to Selected, Unselected, or Required (default is Selected). Together, these settings determine how the package is displayed on the Installer Custom Install pane.

On that pane, a selected checkbox means the package will be installed. If the checkbox is deselected, the package won't be installed. If an entry is dimmed, you can't change the setting. And if an entry is not dimmed, you can select or deselect the package for installation. Table 1 shows how each package is displayed and its resulting status (required or not required), based on the possible settings in the package and metapackage.

Table 1 Relationship between the Required checkbox in the package and the package attribute in the metapackage (Required, Selected, Unselected)

Package	Metapackage	Custom Install checkbox in Installer	Result
Required	Selected	checked + dimmed	required
Required	Unselected	checked + dimmed	required
Required	Required	checked + dimmed	required
not Required	Selected	checked + enabled	not required
not Required	Unselected	not checked + enabled	not required
not Required	Required	checked + disabled	required

Authorization and Restart Action

A metapackage requires authorization if any of its enclosed packages requires it; otherwise, a metapackage does not require authorization. A metapackage requires the highest authorization required by any enclosed package; it also requires the highest restart action of any enclosed package.

Hiding the Packages Contained in a Metapackage

Something to consider when creating a metapackage is that the packages (or metapackages) it contains are visible to users. If you want to discourage users from installing individual packages, which is generally recommended, you can place the contained packages in a directory that isn't visible in Finder windows. You can make a directory invisible (given standard Finder settings) by changing its name to start with a period.

One convenient way to create a metapackage that hides its packages in an invisible directory is to create a visible directory, add the packages to the directory, create and test the metapackage, then make the directory invisible by adding a period to its name using a shell command. After renaming the directory, you'll have to edit the `Info.plist` file in the metapackage to point to the packages.

Suppose, for example, you have two packages, `Cool_App.pkg` and `Mouse_pad.pkg` (the same packages used in "Execution Order for a Complex Metapackage" (page 56)). You could create a metapackage that installs these packages from a hidden directory by performing the following steps:

1. In the Finder, in the directory where you want the metapackage file to reside, create a subdirectory named `contained_packages`. Copy `Cool_App.pkg` and `Mouse_pad.pkg` into that directory.
2. Open PackageMaker and choose File > New > Meta Package to create a new metapackage.
3. In the Finder, drag each of the packages to the Package List pane in PackageMaker.
4. Choose File > Create Package and create the metapackage definition file as `TestHiddenPackage.mpkg`. Save this file in the same directory as `contained_packages`.
5. Choose File > Save and save the metapackage definition file as `TestHiddenPackage` (it will automatically get the extension `.pmsm`). For convenience, save this file in the directory with the metapackage.
6. Test the metapackage. If there are any problems, open the metapackage definition file to modify and save the metapackage, and continue testing until it works as desired.
7. In a Terminal window, use the `cd` command to change directories to the directory containing the metapackage and other files.
8. Use the `mv` command to add a period to the beginning of the `contained_packages` directory:

   ```
   mv contained_packages .contained_packages
   ```

   The period causes the directory to disappear in the Finder unless you have your Finder settings adjusted to show hidden files.
9. In the Finder, Control-click `TestHiddenPackage.mpkg` and choose Show Package Contents.
10. In the Finder or in a Terminal window, set the permissions on the file `Info.plist` in the `Contents` directory so that you have permission to edit it. For example, in a Terminal widow, you could use

    ```
    chmod +w Info.plist
    ```

    After editing, you can use `chmod -w Info.plist` to prevent further modifications.
11. Open the file and look for the key `IFPkgFlagComponentDirectory`. Change the value for that key from ".." to "../.contained_packages".

12. Save your changes to the `Info.plist` file and test the metapackage. It should work the same as before but with the package files no longer visible (again, assuming standard settings in the Finder).

Note: If you know the name of an invisible directory, you can open the directory by doing a Finder search by name, and adding a search criteria for Visibility: all. Then double-click the icon for the found directory to open it.

You can also perform the following steps to show all invisible files in the Finder:

1. Type the following line in a Terminal window and press Return.

```
defaults write com.apple.Finder AppleShowAllFiles YES
```

You can turn off display of invisible files with the following line:

defaults write com.apple.Finder AppleShowAllFiles NO

2. Relaunch the Finder. You can do so by choosing Force Quit from the Apple menu, selecting the Finder, and clicking Relaunch.

For more information on editing property list files, see "Examining and Modifying Property Lists" (page 87).

Authorization, File Ownership, and Permissions

This section describes how permissions are set for files and directories during installation with PackageMaker and Installer. It also describes how to specify authorization requirements, and when the user will be required to authenticate.

File Ownership

By default, if no authorization is required, the files that Installer places on a user's computer are owned by the user doing the installation. If authorization is required, the files are owned by the owner specified in the files archive within the package, regardless of the user and password supplied to complete the authentication. So if your software requires authorization, make sure all of the files and directories have the desired owner and group when you set up a directory structure prior to creating a package. See "Setting File Ownership and Permissions" (page 80) for suggestions on how to do this.

Authorization and authentication are described in "Authorization for an Installation" (page 82).

File Permissions

By default, the files and directories that Installer places on a user's computer have the permissions specified in the package's archive. You can override the default permission settings for installed directories (but not files) by selecting the Overwrite Permissions option in the PackageMaker Info pane. This instructs Installer to use the permissions of directories in the package to modify the permissions of any identically named directories found on the installation volume.

Warning: Use this option with care. If the installation sets the permissions of files in the root directory incorrectly, the installation of the package could render the target computer unusable. For related information, see "Things to Look Out For" (page 84).

To elaborate on how permissions are set for installed files and directories:

- All installed files get their permissions from the package's file archive.
- All installed directories that don't already exist on the target volume get their permissions from the package's file archive.
- All directories that already exist on the target volume keep their current permissions, *unless* the package specifies the Overwrite Permissions option.
- In some cases, setting the desired permissions may require authorization. You can request authorization as described in "Authorization for an Installation" (page 82).

In general, when you set up a directory structure for your software prior to creating a package, make sure all of the permissions are set as you want them when installed. See "Setting File Ownership and Permissions" (page 80) for suggestions on how to do this.

Setting File Ownership and Permissions

For the majority of installations, which install a relocatable application, you can do the following:

1. Set the Default Location (in the PackageMaker Info pane) to `/Applications`.

   Note: It's a good idea to always specify as much directory structure as you can in the default location and as little as possible in your package.

2. Set permissions for the directories and executable files in the package to a value of 775 (full access for owner and group, read and execute access for others).

   Set permissions for non-executable files to 664 (read and write access for owner and group and read-only access for others).

   Note: These are suggestions. In some cases (such as for a daemon or specialized software) you may know that files in a package require other permissions. Whatever permissions you set, be sure to perform test installations with the package before distributing it.

3. Set an appropriate owner and group for the packaged files and directories. You can read more about how to get this information in "Obtaining Owner and Group Information" (page 81).

   For example, applications often are owned by `root` and have a group of `admin`.

   Note: For an application whose owner is displayed as `root` in a Terminal window (as in Listing 1 (page 81)), the owner is shown as "System" instead when you choose File > Get Info in the Finder.

   System software, such as a kernel extension (or KEXT), is owned by `root` and has a group of `wheel`.

   Important: This step is particularly important if your package requires authorization. If so, do not use your own owner and group settings, unless you truly want the installed files on the users system to have those settings. For more information, see "File Ownership Issues Can Cause Problems" (page 85).

4. Set the Authorization Action pop-up to Admin Authorization (also in the PackageMaker Info pane). See Table 1 (page 83) for information about possible settings.

If your installation is not relocatable, you can do the following:

1. Set the Default Location to the full path in which to place the software.
2. Set the required permissions for the installation location.

3. Set an appropriate owner and group for the packaged files and directories, as described in the previous steps.

4. Set the Authorization Action pop-up to Admin Authorization or Root Authorization, as appropriate.

Obtaining Owner and Group Information

"Authorization, File Ownership, and Permissions" (page 79) explains why, if your software requires authorization, you should make sure all of the files and directories have the desired owner and group when you set up a directory structure prior to creating a package. And "File Ownership Issues Can Cause Problems" (page 85) describes what can go wrong if you don't follow that advice. So how do you determine which owner and group to use?

If your software will be installed in a known destination, such as `/Applications`, you can examine the owner and group for files and directories already installed in that location. An easy way to do this is to open a Terminal window and display the information using a command such as the `ls` command. For example, you could type `cd /Applications; ls -l` (and press Return) to change location to the Applications directory and list all the files it contains, along with their permissions, owner, group, and other information.

When you do this, you should see that many files have an owner of `root` and a group of `admin`. These are values you can use for an application you place in that location. Some files and directories may be owned by the current user (which is probably you), but you don't want to be the owner when these files are installed from your package (unless you're packaging software just to install on your own computer).

Similarly, if you list the files in `/System/Library/Frameworks`, you'll see frameworks with owner `root` and group `wheel`.

Sometimes owner, group, and permission information can be damaged or inadvertently changed, so when you check, you may see odd values. To make sure you are seeing the correct values, you can take one of several steps before you check owner and group information:

- Do a clean install of Mac OS X.
- Use the Repair Disk Permissions command in the Disk Utility application (located in `/Applications/Utilties`).

There is also another way to check the owner and group information for a destination, if the directory is part of a system installation—you can examine information from the receipt that was created when Mac OS X was installed. For example, you could use the `lsbom` command (located in `/usr/bin`) to check the `.bom` file in the receipt file `Library/Receipts/BaseSystem.pkg`. You might use the following format for the command:

```
lsbom -p MUGsf /Library/Receipts/BaseSystem.pkg/Contents/Archive.bom
```

Listing 1 shows the first few lines of output from running this command. For information on the `lsbom` flags used in this example, type `man lsbom` in a Terminal widow.

Listing 1 Output from lsbom command

```
drwxrwxr-t  root admin      .
drwxrwxr-x  root admin      ./Developer
```

```
drwxrwxr-x  root admin      ./Developer/Applications
drwxrwxr-x  root admin      ./Developer/Applications/Xcode.app
drwxrwxr-x  root admin      ./Developer/Applications/Xcode.app/Contents
-rw-rw-r--  root admin 13199./Developer/Applications/Xcode.app/Contents/Info.plist
drwxrwxr-x  root admin      ./Developer/Applications/Xcode.app/Contents/MacOS
-rwxrwxr-x  root admin 103072./Developer/Applications/Xcode.app/Contents/MacOS/Xcode
-rw-rw-r--  root admin   8  ./Developer/Applications/Xcode.app/Contents/PkgInfo
drwxrwxr-x  root admin      ./Developer/Applications/Xcode.app/Contents/Plug
```

You can also use the `grep` command to search through all the receipt files in `/Library/Receipts` for packages that have installed into a particular directory. For example, if you wanted to know which receipts recorded an installation into the `/usr/share` directory, you could type the following in Terminal:

```
grep -rs /usr/share /Library/Receipts
```

Listing 2 shows the first few lines of output from running this command.

Listing 2 Output from grep command

```
Binary file /Library/Receipts/AdditionalSpeechVoices.pkg/Contents/Archive.bom matches
Binary file /Library/Receipts/AirPortSW.pkg/Contents/Archive.bom matches
Binary file /Library/Receipts/Apr2004XcodeToolsExtras.pkg/Contents/Archive.bom matches
Binary file /Library/Receipts/AsianLanguagesSupport.pkg/Contents/Archive.bom matches
Binary file /Library/Receipts/BaseSystem.pkg/Contents/Archive.bom matches
Binary file /Library/Receipts/BluetoothUpdate1.5.pkg/Contents/Archive.bom matches
Binary file /Library/Receipts/BluetoothUpdate141.pkg/Contents/Archive.bom matches
```

Authorization for an Installation

Installing a package or metapackage may result in actions that require root or administrator privileges, such as copying files into a `root`-owned directory or copying files that have root permission.

- If the user performing the installation has the required privileges, no authorization is necessary.
- If the user does not have the required privileges and the package is not set up to request authorization, the installation fails and displays a dialog that it cannot proceed.
- If the user does not have the required privileges and the package *is* set up to request authorization, Installer displays an authentication dialog as the first step of the installation.

Note: Authorization is the act of granting a right or privilege, and authentication is the act of verifying the identity of the user. Authorization uses authentication as part of its process.

Recall that if no authorization is required, the files that Installer places on a user's computer are owned by the user doing the installation. If authorization *is* required, the files are owned by the owner specified in the files archive within the package, regardless of the user and password supplied to complete the authentication.

You specify an authorization level for a package using the Authorization Action pop-up menu in the PackageMaker Info pane. The available choices are Root Authorization, Admin Authorization, and No Authorization Required. The user is prompted for authorization (through an authentication dialog) only when the requested authorization is higher than the user's current authorization at installation time. Table 1 shows the possible permissions, settings, and outcomes.

Important: Even if a package specifies root authorization, a user can authenticate by supplying the administrator password and can then install the software.

Table 1 Possible permissions, authorization action settings, and results

User logged in as	Authorization action specified by package	Privileges Used	Result
Root	(any)	Root	Authorization not required
Administrator	Root Authorization	Root	Prompt for authorization
Administrator	Admin Authorization	Administrator	Authorization not required
Administrator	No Authorization Required	Administrator	Authorization not required
Regular user	Root Authorization	Root	Prompt for authorization
Regular user	Admin Authorization	Administrator	Prompt for authorization
Regular user	No Authorization Required	Logged-in user	Authorization not required

A metapackage requires authorization if any of its enclosed packages requires it; otherwise, a metapackage does not require authorization. A metapackage requires the highest authorization required by any enclosed package; it also requires the highest restart action of any enclosed package.

Choosing an Authorization Setting

In general, choose an authorization setting that is the same or higher than the permissions of the files in the package and the specified destination for installing them. Authorization should be set to `root` if any component needs to be set to `root`.

In specific situations, your package should request root or administrator privileges:

- The package contains files that will be installed in a system-level location, such as `/System`, `/usr`, `/bin`, and so on; for example, KEXTs (kernel extensions) reside in `/System/Library/Extensions`.
- The package contains files that will be installed with root or administrator permissions; see "Setting File Ownership and Permissions" (page 80) for related information.

If your package is relocatable, a user can attempt to install the software in any location (unless you've also specified Root Volume Only). Therefore, set an Authorization level that makes sense for your software. (Relocatable and Root Volume Only are options you set in the PackageMaker Info pane.)

Note: It is not recommended that you use the scripts described in "Modifying an Installation With Scripts" (page 139) to modify file permissions.

Things to Look Out For

The following are some installation issues to be aware of.

Installing Can Change Permissions of a User's Directories

Suppose you want to create a package to install a utility in `/Applications/Utilities`, a directory that typically exists on all boot volumes. Further, suppose that in creating the package, the utility application resides in the directory `Package_contents/Applications/Utilities`, which has permissions settings of 750 (full access for owner, read and execute access for group, and no access for others), and you specify a default installation location of `/`.

If you select the Overwrite Permissions checkbox in the PackageMaker Info pane, Installer uses the permissions of directories in the package to modify the permissions of any identically named directories found on the installation volume. This may result in an error if your package does not require authorization (see "Restrictive Permissions Can Cause an Installation to Fail" (page 83).

Even if your packages does require authorization, you may still have a problem. After installation, the user's `/Applications/Utilities` directory has permissions of 750. If the directory previously had a more (or less) permissive setting, the user may be confused and unhappy when the directory can no longer be accessed in the same way as before.

Note: The Overwrite Permissions checkbox might more accurately be titled Overwrite Existing Directory Permissions, because it affects only directories, not files.

This suggests you should select the Overwrite Permissions checkbox only if you are certain that you know better than the user what the directory permissions should be. One such example might be if you are doing net installations in an educational environment.

Installing Can Convert Symbolic Links to Actual Directories

Again, suppose your package installs a utility application in `/Applications/Utilities`, a directory that typically exists on all boot volumes. In this scenario, suppose the user has replaced `Utilities` with a symbolic link (alias) to a `Utilities` directory on a different drive.

By default, Installer replaces links with directories, which again is likely to confuse (and annoy) the user, since it will "orphan" whatever the symbolic link points to. In this case, the user would end up with a `Utilities` directory containing just your utility, while their original `Utilities` directory has been set adrift.

PackageMaker currently has no user interface option to control how Installer handles symbolic links. However, there is a way you can change the default behavior so that Installer follows symbolic directory links, rather than replaces them. To do so, you edit the `Info.plist` file for the package and set the value for the `IFPkgFlagFollowLinks` key to Yes. For more information on making these kinds of changes, see "Examining and Modifying Property Lists" (page 87).

Restrictive Permissions Can Cause an Installation to Fail

Suppose you set the permissions to a restrictive value. If you do not set the Authorization Action pop-up (in the PackageMaker Info pane) to require root or administrator authorization for the package, the installation may fail for a user who isn't logged in with high enough permissions.

File Ownership Issues Can Cause Problems

Suppose your package installs a framework in `/System/Library/Frameworks`, which has an owner of `root`. Suppose the `Frameworks` directory you use to create your package has you (`jbob3`) as the owner. You set the package to request root authorization so that it can be installed in the `Frameworks` directory.

Because the user is authorized as `root`, the Installer is also `root` and can install files using owner `jbob3`. This translates to some user ID number that is likely to be of little value to the current user. And in fact, the user may experience serious problems, as the system may not be able to function properly because it doesn't own files it expects to own.

You can minimize problems of this nature by following the advice provided in "Setting File Ownership and Permissions" (page 80).

Examining and Modifying Property Lists

This section describes how to examine and directly modify the property lists that are used to store installation information in a package. It also provides reference for the Installer keys that you can add or modify.

For additional information on Installer keys, see "Specifying Installation Requirements" (page 111).

Installer Keys

When you create a package with PackageMaker, you use the PackageMaker user interface to supply information that is then stored in property list files in the package. PackageMaker creates two kinds of property lists, described in "Installer Keys" (page 87) and "The PackageMaker Description Property List" (page 95). For information on additional property lists you may need to modify, see "Property Lists You Can Create" (page 96).

The information in a property list is stored as a series of keys and associated values. For example, if your package does not require user authorization, the `Info.plist` file will contain the key `IFPkgFlagAuthorizationAction` with the string value "NoAuthorization". The "IF" in the key name stands for Installation framework. Listing 1 shows the property list for a simple package.

Listing 1 The property list for a simple package

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>IFMajorVersion</key>
    <integer>0</integer>
    <key>IFMinorVersion</key>
    <integer>0</integer>
    <key>IFPkgFlagAllowBackRev</key>
    <false/>
    <key>IFPkgFlagAuthorizationAction</key>
    <string>NoAuthorization</string>
    <key>IFPkgFlagDefaultLocation</key>
    <string>/</string>
    <key>IFPkgFlagInstallFat</key>
    <false/>
    <key>IFPkgFlagIsRequired</key>
    <false/>
    <key>IFPkgFlagOverwritePermissions</key>
    <true/>
    <key>IFPkgFlagRelocatable</key>
    <false/>
    <key>IFPkgFlagRestartAction</key>
    <string>NoRestart</string>
```

```
    <key>IFPkgFlagRootVolumeOnly</key>
    <false/>
    <key>IFPkgFlagUpdateInstalledLanguages</key>
    <false/>
    <key>IFPkgFlagUseUserMask</key>
    <false/>
    <key>IFPkgFormatVersion</key>
    <real>0.10000000149011612</real>
</dict>
</plist>
```

Editing a Property List

Some developers will never need to directly edit a package's information property list or description property list, but there are several reasons why you might. These reasons are described in "The Package as a Black Box" (page 32). For example, you might need to use a feature that is not accessible through the PackageMaker user interface.

"Other Keys You Might Want to Modify" (page 96) lists examples of several property list keys you might want to directly modify. For an example that provides step-by-step instructions for editing the property list file of a metapackage, see "Hiding the Packages Contained in a Metapackage" (page 77).

You can examine the contents of a package in the Finder by Control-clicking it and choosing Show Package Contents from the contextual menu. You can then view or edit the `Info.plist` and `Description.plist` files in an appropriate XML or text editor. You may have to change the file's permissions before editing one of these files.

Note: Double-clicking the `Info.plist` file opens it in Property List Editor, but due to a bug in some versions prior to the one released with Mac OS X version 10.3, Property List Editor may not display the keys correctly in its table view. It does howevever display the underlying text correctly in its text view (click the Dump button).

You can also examine the contents of a package in a Terminal window using the `find` command. It's also possible to directly edit a property list file as you would any XML file. Listing 2 shows the result of using the `find` command to display the contents of a package in the current directory. The types of files contained in a package are described in "Anatomy of a Package" (page 69).

Warning: If you open a package definition file (a file with extension `.pmsp`) in PackageMaker and recreate its package with the same name and location, you overwrite any changes you may have made to the `Info.plist` file or other package files created by PackageMaker. Therefore, if you switch back and forth between direct editing and editing with PackageMaker, be sure to save copies of your edited files and reinsert your changes.

Listing 2 Displaying the contents of a package in a Terminal window

```
% find Simple.pkg
Simple.pkg
Simple.pkg/Contents
Simple.pkg/Contents/Archive.bom
Simple.pkg/Contents/Archive.pax.gz
```

```
Simple.pkg/Contents/Info.plist
Simple.pkg/Contents/PkgInfo
Simple.pkg/Contents/Resources
Simple.pkg/Contents/Resources/Description.plist
Simple.pkg/Contents/Resources/Simple.bom
Simple.pkg/Contents/Resources/Simple.info
Simple.pkg/Contents/Resources/Simple.pax.gz
Simple.pkg/Contents/Resources/Simple.sizes
```

The PackageMaker Information Property List

An information property list contains key-value pairs that specify various information about an application or bundle that can be used at runtime. When you create a package, PackageMaker stores much of the information you supply in the package's `Info.plist` file. Each package has one such file, stored in the `Contents` directory. This file is not localizable.

You're most likely to be interested in Installer key information if you want to modify a package directly by editing its property list files. The Installer keys used in the `Info.plist` file are documented in the following sections:

- "Keys That Can Appear in Any Package" (page 89)
- "Keys That Can Appear Only in Single Packages" (page 91)
- "Keys That Can Appear Only in Metapackages" (page 94)

Some keys are not actually defined by Installer—for example, `CFBundleName` and `CFBundleIdentifier` are standard bundle keys that are used by many other applications. However, as described here, they are used in a package property list to provide installation information.

Important: For clarity, sample values for string keys are shown in quotation marks. However, the actual value does not include the quotation marks.

Keys That Can Appear in Any Package

The following keys, listed in alphabetical order, can be used in packages or metapackages:

`CFBundleGetInfoString`

optional

data type: string

default value: none

example: "My Great Application 1.0"

Provides information for the Finder to display about the package.

`CFBundleIdentifier`

required

data type: string

default value: none

example: “com.mycompany.mygreatapplication”

A unique identifier string for the bundle. It should be in the form of a Java-style package name. For more information, see “Property List Key Reference” in *Runtime Configuration Guidelines*.

`CFBundleName`

optional

data type: string

default value: none

example: “My Great Application”

Provides information for the Finder to display about the package.

`CFBundleShortVersionString`

optional

data type: string

default value: none

example: “1.0”

Provides information for the Finder to display about the package.

`IFMajorVersion`

optional

data type: int

default value: none

example: 1

Although some versions of Package Maker allow you to enter a value for this key in the “Major” field, and you may see it in the information property list for a package, this key is not used by Installer.

`IFMinorVersion`

optional

data type: int

default value: none

example: 0

Although some versions of Package Maker allow you to enter a value for this key in the “Minor” field, and you may see it in the information property list for a package, this key is not used by Installer.

`IFPkgFlagBackgroundAlignment`

optional

data type: string

values: “left”, “right”, “top”, “bottom”, “center”, “topleft”, “topright”, “bottomleft”, “bottomright”

default value: “center”

example: “bottom”

Specifies the alignment for a background image. Most useful for an image that is smaller than the installer window. Background images and alignment are described in “Supplying a Background Image” (page 64).

`IFPkgFlagBackgroundScaling`

optional

data type: string

values: “none”, “tofit”, “proportional”

default value: “tofit”

example: “proportional”

Specifies the scaling for a background image. Most useful for an image that is larger, in at least one dimension, than the installer window. Background images and scaling are described in “Supplying a Background Image” (page 64).

`IFPkgFormatVersion`

required

data type: float

default value: none

example: 0.10000000149011612

Used internally by Installer to determine the package format and version. If missing, or if value is not greater than 0.1, the package is not read correctly, and fails to load.

You should not change this key.

Keys That Can Appear Only in Single Packages

The following keys, listed in alphabetical order, can be used only in packages:

`IFPkgFlagAllowBackRev`

optional

data type: BOOL

default value: No

example: Yes

Should Installer allow reverting to an earlier version of the package?

`IFPkgFlagAuthorizationAction`

optional

data type: enum

values: NoAuthorization, AdminAuthorization, RootAuthorization

default value: NoAuthorization

example: AdminAuthorization

Should Installer require authorization (and prompt the user to authenticate)?

For related information, see Table 1 (page 83).

`IFPkgFlagDefaultLocation`

optional

data type: string

default value: "/"

example: "/Applications"

The initial install location. If the package is relocatable, the user can change the install location.

See also `IFPkgFlagRelocatable`.

For related information, see "Specify the Installation Destination" (page 47).

`IFPkgFlagFollowLinks`

optional

data type: BOOL

default value: No

example: Yes

If a symbolic link is found on the target volume for a directory listed in the package, the link is resolved and the contents are replaced at the link location. Otherwise, the symbolic link is replaced by a similarly named directory, breaking the user's link.

For more information, see "Installing Can Convert Symbolic Links to Actual Directories" (page 84).

`IFPkgFlagInstallFat`

optional

data type: BOOL

default value: No

example: Yes

If the binary contains code for multiple platforms, should Installer install all of them? (If not, Installer strips out all but the code necessary for the current platform.)

Important: If you are installing `.o` or `.a` files as part of your package, they will be thinned along with any other binaries. To prevent this outcome, set the value for this key to Yes.

`IFPkgFlagIsRequired`

optional

data type: BOOL

default value: No

example: Yes

Is this package a required component for the installation?

`IFPkgFlagOverwritePermissions`

optional

data type: BOOL

default value: No

example: Yes

Should the permissions of directories in the package override those of any directories with the same names found on the installation volume?

`IFPkgFlagRelocatable`

optional

data type: BOOL

default value: No

example: Yes

Should the user be allowed to choose a destination for the installation?

See also `IFPkgFlagDefaultLocation`.

For related information, see "Specify the Installation Destination" (page 47).

`IFPkgFlagRestartAction`

optional

data type: enum

values: NoRestart, RecommendedRestart, RequiredRestart, Shutdown, RequiredLogout

default value: NoRestart

example: RequiredRestart

Should Installer require or recommend a restart, logout, or shutdown? (Only takes place if user installs on currently booted volume.)

For related information, see "Restarting, Logging Out, or Shutting Down" (page 39) and "Other Keys You Might Want to Modify" (page 96).

`IFPkgFlagRootVolumeOnly`

optional

data type: BOOL

default value: No

example: Yes

Should Installer limit the installation destination to the boot volume?

`IFPkgFlagUpdateInstalledLanguages`

optional

data type: BOOL

default value: No

example: Yes

Should Installer limit updating of current languages?

Keys That Can Appear Only in Metapackages

The following keys, listed in alphabetical order, can be used only in metapackages.

`IFPkgFlagComponentDirectory`

required

data type: string

default value: ".."

example: "../.contained_packages"

Specifies the location of the contained packages or metapackages, as a path relative to the containing metapackage. Prepended to the name field of each package or metapackage to locate it.

The building of paths to subpackages for a metapackage starts with the path to the metapackage. The component directory is then added. The use of ".." specifies one directory up from the current path, which is the directory containing the metapackage. So the default option is to look for contained packages at the same level as the metapackage itself.

Setting the value of this key to "../.contained_packages" would specify a directory named `contained_packages` within the directory containing the metapackage.

For an example that modifies this key, see "Hiding the Packages Contained in a Metapackage" (page 77).

`IFPkgFlagPackageList`

required

data type: array of dictionaries

default value: none

Specifies the contained packages or metapackages. Array includes a dictionary for each package or metapackage; each dictionary has the following keys: `IFPkgFlagPackageLocation` and `IFPkgFlagPackageSelection`.

Since a metapackage cannot be created without at least one package in the package list, this array will be filled in and the dictionaries for each package will be built as well.

`IFPkgFlagPackageLocation`

required

data type: string

default value: none

example: "MyPackage.pkg"

The full name of the contained package or metapackage. Must have the extension `.pkg` or `.mpkg`.

Used as a key in a dictionary defined as part of the `IFPkgFlagPackageList` key.

`IFPkgFlagPackageSelection`

optional

data type: enum

values: selected, unselected, required

default value: selected

example: required

Used in determining whether checkbox for installing the package or metapackage is shown as selected or not, and whether it is enabled. For more information, see "When a Package or Metapackage Is Required" (page 76).

Used as a key in a dictionary defined as part of the `IFPkgFlagPackageList` key.

The PackageMaker Description Property List

When you create a package, some of the information you supply in PackageMaker winds up in the description property list file (`Description.plist`). This file is similar to the `Info.plist` file, in that both contain key-value pairs that store information about the package. However, the description property list contains information that is localizable (currently gathered from the PackageMaker Description pane). PackageMaker provides a Description pane for both packages and metapackages, and both types of package have `Description.plist` files.

By default, the `Description.plist` file is located in the package's `Resources` directory, but it may instead be in one of the language project subdirectories, such as `French.lproj`. Currently, Installer uses the value of the `IFPkgDescriptionTitle` key as part of the title of the installer window, and on the Welcome page as well. It uses the value of the `IFPkgDescriptionDescription` key in the Custom Install pane (available only to metapackages) when a package or metapackage is selected.

Important: For clarity, sample values for string keys are shown in quotation marks. However, the actual value does not include the quotation marks.

Description Property List Keys

The following keys, listed in alphabetical order, can be used in all packages.

`IFPkgDescriptionTitle`

required

data type: string

default value: none

example: "My Excellent Software"

The name of the package or metapackage, or other short, descriptive information about it.

`IFPkgDescriptionVersion`

optional

data type: string

default value: none

example: "1.1"

Specifies the version for the package or metapackage. Not currently used.

Property Lists You Can Create

You can provide information to help Installer find your previously installed software during future upgrades. If you do, you'll need to create a property list named `TokenDefinitions.plist` to specify information about the search methods to employ.

For information about the structure and keys used in this property list, see "Finding Previously Installed Software" (page 97). For another situation in which you might work with this list, see the section "Bundle Requirements" (page 121) in "Specifying Installation Requirements" (page 111).

Other Keys You Might Want to Modify

There are several situations where you may need to modify a property list key as part of specifying a feature.

- Starting in Mac OS X version 10.3, you can provide information to specify requirements for the installation of your software, as described in "Specifying Installation Requirements" (page 111).
- In providing information to help Installer find your previously installed software, you supply various keys and values, as described in "Finding Previously Installed Software" (page 97).
- You specify alignment and scaling for a background image by setting the `IFPkgFlagBackgroundAlignment` and `IfPkgFlagBackgroundScaling` keys, respectively, as described in "Supplying a Background Image" (page 64).
- To hide the package files in a metapackage, you may need to set the value of the `IFPkgFlagComponentDirectory` key, as described in "Hiding the Packages Contained in a Metapackage" (page 77).
- You can change the default behavior of the Installer application in treating symbolic links by setting the value of the `IFPkgFlagFollowLinks` key, as described in "Installing Can Convert Symbolic Links to Actual Directories" (page 84).
- Starting in Mac OS X version 10.3, you can specify that an installation requires a logout by setting the value of the `IFPkgFlagRestartAction` key in the package's information property list to `RequiredLogout`. For information on other restart actions, see "Restarting, Logging Out, or Shutting Down" (page 39).

Finding Previously Installed Software

This section describes the Find File feature, which allows you to provide Installer with information to help it find software from a previous installation. This feature was introduced in Mac OS X version 10.2. Enhancements introduced in Mac OS X version 10.3 are noted throughout this section.

Installer looks for a receipt in `/Library/Receipts` that matches the current package to determine whether you are installing software for the first time or upgrading it. However, the receipt doesn't necessarily specify where the software currently resides. If Installer can't find previously installed software, it can't upgrade it, and instead must do a full installation.

Note: If a receipt exists, Installer reports that it is doing an upgrade, even if it doesn't find the previously installed software and must do a full installation. For more information, see "Installs and Upgrades" (page 33).

You can use the Find File feature to provide rules to help Installer find software from a previous installation. This feature is most useful for software that can easily be moved by the user after installation, such as self-contained application bundles. It isn't needed for any software that must reside in a certain location, such as a KEXT (kernel extension). Application bundles are described in "Drag-and-Drop Installation" (page 18).

In conjunction with using the Find File feature to help Installer find previously installed software, you can use the feature described in "Specifying Version Information for Packaged Software" (page 107) to help Installer determine which software needs to be upgraded.

PackageMaker does not currently provide a user interface for accessing the Find File feature. As a result, to use it, you supply information by directly editing the information in property list files in a package. The information you need to supply is described below; for information on how to edit property list files, see "Examining and Modifying Property Lists" (page 87).

Tokens Definitions and Path Mappings

To help Installer find your previously installed software, you provide information about the software items to search for and the search methods to use. To do this, you create token definitions and path mappings in your package.

Token Definitions

A token acts as an identifier for an abstract place in the file system. A token may resolve to different locations depending on how the user has arranged his or her files. For example, you could define a token "iChat" whose value is the path to the `iChat.app` application bundle, wherever the user has placed it.

To define tokens within your package, you create a property list file named `TokenDefinitions.plist` in your package's `Resources` directory. The top-level container in this file should be a dictionary, and each key should be the name of a token that you wish to define in your package. The value for each token should be an array of one or more invocations to search methods, specifying the search logic to use when resolving the token. For more detailed information on creating token definitions, including documentation of the specific search methods, see "Search Methods" (page 100).

Listing 1 shows a sample `TokenDefinitions.plist` file in XML format. This property list defines a token "ClockToken" corresponding to `Clock.app`, which is typically located in `/Applications`. This example uses the search method `CommonAppSearch`, which is specified as the value for the `searchPlugin` key. For information on this search method, see "CommonAppSearch" (page 104).

Listing 1 A TokenDefinitions.plist file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
    <dict>
        <key>ClockToken</key>
        <array>
            <dict>
                <key>searchPlugin</key>
                <string>CommonAppSearch</string>
                <key>path</key>
                <string>/Applications/Clock.app</string>
                <key>identifier</key>
                <string>com.apple.clock</string>
            </dict>
        </array>
    </dict>
</plist>
```

Path Mappings

A path mapping entry tells Installer which files within your package should be installed at the location defined by the token, rather than at the default location where the package would otherwise install them. That is, the token is resolved according to the search method or methods it specifies, and the result is used to specify the installation location.

A path mapping specifies the beginning part of a path, and every file within the package whose path begins with the specified path beginning will be installed at the new location.

If you don't specify a token and a path mapping, or if you specify them but the software to be updated is not found at the location they specify, files in your package are installed at a location that is constructed from the installation volume chosen by the user, the default location specified in your package, and the relative path to the file in the package (which is based on how you set up the files relative to your root directory when you created the package).

Note: During testing, you can see where your files are actually installed by looking in the Installer Log window, which is described in "Viewing the Installer Log and Script Output" (page 50).

If you do specify a token (with one or more search methods) and a path mapping, and the software to be updated is found, the token is resolved to the path to the location where the software was found. This path includes volume information—a token may resolve to a location on a different volume than the installation volume chosen by the user. Files that match the path mapping are then installed at a location constructed from the resolved path (including volume), plus the part of the path in the package that wasn't matched by the path mapping.

To add path mappings, you create a new key `IFPkgPathMappings` in your package's `Info.plist` file. The value of this property is a dictionary where each key is a partial path inside the package. These paths should correspond to the relative paths output by the `lsbom` tool. The value for each entry is a string that specifies the new path that a file should be mapped to. The path can begin with a symbolic reference to a token, surrounded by curly braces—for example, `{ClockToken}`.

Note: For related information, see the documentation for IFPkgFlagDefaultLocation in "Keys That Can Appear Only in Single Packages" (page 91).

Listing 2 shows an entry for the `IFPkgPathMappings` key from a package's `Info.plist` file. This mapping causes the `Clock.app` bundle and all of its contained files to be installed in the location resolved by the token defined in Listing 1.

Listing 2 An entry for the IFPkgPathMappings key in an information property list

```
<key>IFPkgPathMappings</key>
<dict>
    <key>./Applications/Clock.app</key>
    <string>{ClockToken}</string>
</dict>
```

Application Selection By the User

Starting in Mac OS X version 10.3, a package can specify that the user should be consulted regarding which version to update if multiple copies are found. This is accomplished by defining a special token named `UserLocation`. The `UserLocation` token is resolved like any other token, but if multiple copies of the specified software are found, the user is asked to choose which copy to upgrade. The value that the user chooses becomes the new value of the token when it is used in path mappings (described in "Path Mappings" (page 98).

Listing 3 shows a `UserLocation` token definition in a `TokenDefinitions.plist` file (described in "Token Definitions" (page 97)).

Listing 3 A UserLocation token in a TokenDefinitions.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
    <dict>
```

```
        <key>UserLocation</key>
        <array>
            <dict>
                <key>searchPlugin</key>
                <string>CommonAppSearch</string>
                <key>path</key>
                <string>/Applications/Clock.app</string>
                <key>identifier</key>
                <string>com.apple.clock</key>
            </dict>
        </array>
    </dict>
</plist>
```

Search Methods

You can use token definitions to create highly customized behaviors for locating and identifying your software. You do this by stringing together search method invocations. The search methods are invoked in the order specified in the token definition, until one of them returns a success code. At that time, the array of search results is returned to Installer. The array contains the path to the location to which the token resolves.

Important: A user may have versions of the specified software in more than one of the locations you tell Installer to search. Because Installer always updates the first version it finds, you should attempt to set up your searches so that the version you would most like to update is found first. The "CommonAppSearch" (page 104) method attempts to find the version that is most likely to be the one the user wants to update.

To define a token that specifies how to locate your software, you add a key for the token, with an appropriate token name, to the `TokenDefinitions.plist` file. You then add an array of dictionaries as the value for that key. Each dictionary specifies a search method that is invoked during installation. Listing 1 shows a token definition that specifies a single search method.

The following sections describe the search methods Installer currently supports and the required and optional keys for those search methods.

CheckPath

You use the `CheckPath` search method to specify the path where installed software might be found. This is the simplest search method—it succeeds if a file or directory exists at the specified path, and fails otherwise.

The `CheckPath` search method has one key:

`path`

required

A string specifying the path to search (such as `/Applications`).

Listing 4 shows a token definition that uses `CheckPath` to sequentially check several locations where the application may reside.

Listing 4 A token definition that uses the CheckPath search method

```
<key>ClockToken</key>
<array>
    <dict>
        <key>searchPlugin</key>
        <string>CheckPath</string>
        <key>path</key>
        <string>/Applications/Clock.app</string>
    </dict>
    <dict>
        <key>searchPlugin</key>
        <string>CheckPath</string>
        <key>path</key>
        <string>/Applications/Utilities/Clock.app</string>
    </dict>
    <dict>
        <key>searchPlugin</key>
        <string>CheckPath</string>
        <key>path</key>
        <string>~/Applications/Clock.app</string>
    </dict>
</array>
```

LaunchServicesLookup

You use the `LaunchServicesLookup` search method to consult the database maintained by Launch Services in looking for the software.

Note: Launch Services is a framework that provides an API for launching applications in Mac OS X. A framework is a type of bundle (or directory in the file system) that packages software with the resources that software requires.

The `LaunchServicesLookup` search method has the following keys:

`identifier`

optional

The `CFBundleIdentifier` for the bundle the search is intended to locate A `CFBundleIdentifier` is the unique identifier string for a bundle and should be in the form of a java package name (for example, `com.apple.clock`).

Prior to Mac OS X version 10.3 (v10.3), this key was required. Starting in Mac OS X v10.3, you must specify either this key or the `creator` key.

`creator`

optional

A string that specifies a four-character creator (for example, `ttxt` for the TextEdit application).

Prior to Mac OS X v10.3, this key was required. Starting in Mac OS X v10.3, you must specify either this key or the `identifier` key.

`type`

optional

A string that specifies a four-character document type the application can open (for example, `TEXT`).

Starting in Mac OS X v10.3, this key is ignored, and you must specify either the `creator` key or the `identifier` key.

`extension`

optional

The extension for a file type the application can open (for example, `.txt`).

Starting in Mac OS X v10.3, this key is ignored, and you must specify either the `creator` key or the `identifier` key.

Listing 5 shows a token definition that uses `LaunchServicesLookup`.

Listing 5 A token definition that uses the LaunchServicesLookup search method

```
<key>ClockToken</key>
<array>
    <dict>
        <key>searchPlugin</key>
        <string>LaunchServicesLookup</string>
        <key>identifier</key>
        <string>com.apple.clock</string>
    </dict>
</array>
```

BundleIdentifierSearch

The `BundleIdentifierSearch` search method performs an exhaustive search of the entire file system, looking for bundles with the specified `CFBundleIdentifier`.

The `BundleIdentifierSearch` search method has the following required and optional keys:

`startingPoint`

optional

A string specifying the path to from which to start searching. If this is not specified, `/` is assumed.

`identifier`

required

The `CFBundleIdentifier` for the bundle that the search is intended to locate (for example, `com.apple.clock`).

`successCase`

optional

Specifies the behavior to use when the `BundleIdentifierSearch` search method succeeds. It can take one of three values:

- `findOne` Terminates the search as soon as a match is identified. This can be the most efficient choice. It is also the default choice.
- `findAll` Exhaustively iterates over the entire file system, looking for all matches. Succeeds if any matches are found.

- `neverSucceed` Iterates over the entire file system, and compiles results that can be used by the "BundleVersionFilter" (page 103) search method. See Listing 7 for an example.

`maxDepth`

optional

The maximum directory depth to descend in the search. For example, specifying a depth of 3 would limit the search to no more than three directories deep. Provided for efficiency, so that searching needn't proceed down deeply nested hierarchies that are unlikely to contain the software.

`excludedDirs`

optional

An array of one or more strings, each of which identifies a directory that should be excluded from the search. This key is also provided for efficiency: for example, you can avoid a great deal of searching by excluding the locations `/System` and `/Developer` if your software is unlikely to be located within either of those directories.

Listing 6 shows a token definition that uses `BundleIdentifierSearch`.

Listing 6 A token definition that uses the BundleIdentifierSearch search method

```
<key>ClockToken</key>
<array>
    <dict>
        <key>searchPlugin</key>
        <string>BundleIdentifierSearch</string>
        <key>identifier</key>
        <string>com.apple.clock</string>
        <key>excludedDirs</key>
        <array>
            <string>/System</string>
            <string>/Developer</string>
            <string>/AppleInternal</string>
        </array>
        <key>maxDepth</key>
        <integer>6</integer>
        <key>startingPoint</key>
        <string>/</string>
        <key>successCase</key>
        <string>findOne</string>
    </dict>
</array>
```

BundleVersionFilter

You use the `BundleVersionFilter` search method to filter the results returned by the previous invocation of a search method. It succeeds if one or more bundles fall within the specified version range, and fails otherwise. This search method was added in Mac OS X version 10.3.

The `BundleVersionFilter` search method has the following keys (you must specify at least one of the keys):

`maxVersion`

optional

The maximum version allowed. Specified as a 5-tuple version in a string. The first three numbers correspond to the `CFBundleShortVersionString`. For more information, see "How Installer Computes a Version" (page 109).

`minVersion`

optional

The minimum version allowed.

Listing 7 shows a token definition that uses `BundleVersionFilter`.

Listing 7 A token definition that uses the BundleVersionFilter search method

```
<key>ClockToken</key>
<array>
    <dict>
        <key>searchPlugin</key>
        <string>BundleIdentifierSearch</string>
        <key>identifier</key>
        <string>com.apple.clock</string>
        <key>maxDepth</key>
        <integer>6</integer>
        <key>successCase</key>
        <string>neverSucceed</string>
    </dict>
    <dict>
        <key>searchPlugin</key>
        <string>BundleVersionFilter</string>
        <key>maxVersion</key>
        <string>1.0.0.99999999.9</string>
        <key>minVersion</key>
        <string>1.0.0.0.0</string>
    </dict>
</array>
```

CommonAppSearch

If you do not require a highly customized search behavior, you can use the `CommonAppSearch` search method and save a lot of typing. This search method, which is first available in Mac OS X version 10.3, attempts to locate at least one copy of the specified application.

Note: `CommonAppSearch` is simply a macro that expands to a more complicated token definition.

The `CommonAppSearch` search method has the following required keys:

`path`

required

The default path to the application to locate.

`identifier`

required

The `CFBundleIdentifier` of the bundled application to locate.

Listing 1 (page 98) provides an example of how to use `CommonAppSearch`.

A Token Definition for a Complex Search

Listing 8 provides an example of a token definition that combines three search methods: `CheckPath`, `LaunchServicesLookup`, and `BundleIdentifierSearch`.

Listing 8 Expansion of the CommonAppSearch macro

```
<key>ClockToken</key>
<array>
    <dict>
        <key>searchPlugin</key>
        <string>CheckPath</string>
        <key>path</key>
        <string>{path Argument}</string>
    </dict>
    <dict>
        <key>searchPlugin</key>
        <string>LaunchServicesLookup</string>
        <key>identifier</key>
        <string>{identifier Argument}</string>
    </dict>
    <dict>
        <key>searchPlugin</key>
        <string>BundleIdentifierSearch</string>
        <key>identifier</key>
        <string>{identifier Argument}</string>
        <key>excludedDirs</key>
        <array>
                <string>/System</string>
                <string>/Developer</string>
                <string>/AppleInternal</string>
        </array>
        <key>maxDepth</key>
        <integer>6</integer>
        <key>startingPoint</key>
        <string>/</string>
        <key>successCase</key>
        <string>findOne</string>
    </dict>
</array>
```

Specifying Version Information for Packaged Software

If you supply version information in the correct format as part of bundled software you create, PackageMaker can gather and store the information in your package file, and Installer can use it to make intelligent decisions about when to replace outdated software during an installation.

This feature provides Installer with an efficient mechanism for determining which files in a package need to be installed, and can lead to a faster installation. This mechanism works as follows:

- You supply version information in a version property list file named `version.plist` in any application, framework, or other bundles in your installation package.
- When you create a package, PackageMaker automatically searches for version property list files in your software, combines them into another property list named `BundleVersions.plist`, and stores that list in the package.
- During an upgrade installation, Installer can compare version property list files in the previously installed software with those in its bundle versions list. If the version in the target is newer than the version in the package, Installer skips the step of installing that software.

The following sections describe this process, and the bundle versions and version property lists, in more detail.

Note: See "Installs and Upgrades" (page 33) for information on how Installer determines whether an installation is an upgrade or an install.

Supplying Version Information for Your Software

In addition to the version information you typically supply in any bundled software, you can add a small property list file that provides Installer with version information about your software. This file can be stored anywhere in the bundle, must have the name `version.plist`, and should contain information similar to that shown in Listing 1. Of the keys shown in Listing 1, the most important to supply are `BuildVersion`, `CFBundleShortVersionString`, and `SourceVersion`.

Listing 1 Contents of a version.plist file for a beta release of PackageMaker

```
<dict>
    <key>BuildVersion</key>
    <string>17</string>
    <key>CFBundleShortVersionString</key>
    <string>1.1</string>
    <key>CFBundleVersion</key>
    <string>1.1</string>
    <key>ProjectName</key>
    <string>PackageMaker</string>
    <key>ReleaseStatus</key>
```

```
        <string>Beta</string>
        <key>SourceVersion</key>
        <string>200000</string>
    </dict>
```

The BundleVersions.plist File

The `BundleVersions.plist` file (or bundle versions file) is a property list file created by PackageMaker when it creates an installation package. The file contains the contents of any version property list files found by scanning the items contained in the archive of files to be installed.The bundle versions file is stored in a package's `Resources` directory and is not localized.

The bundle versions file contains a dictionary of dictionaries. For each top-level dictionary, the key is the path to the `version.plist` corresponding to one of the bundles being installed. The value of this key is the contents of the `version.plist` referenced by the path in the key.

For example, suppose that there exists an installation package for the latest version of ColorSync Utility and CPU Monitor. The bundle versions file might look like the one shown in Listing 2.

Listing 2 A BundleVersions.plist with versions for two bundles

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist SYSTEM "file://localhost/System/Library/DTDs/PropertyList.dtd">
<plist version="0.9">
<dict>
    <key>./Applications/Utilities/ColorSync Utility.app/Contents/version.plist</key>
    <dict>
        <key>BuildVersion</key>
        <string>19</string>
        <key>CFBundleShortVersionString</key>
        <string>4.0</string>
        <key>CFBundleVersion</key>
        <string>4.0</string>
        <key>ProjectName</key>
        <string>ColorSyncEtc</string>
        <key>ReleaseStatus</key>
        <string>GM</string>
        <key>SourceVersion</key>
        <string>81</string>
    </dict>
    <key>./Applications/Utilities/CPU Monitor.app/Contents/version.plist</key>
    <dict>
        <key>BuildVersion</key>
        <string>35</string>
        <key>CFBundleShortVersionString</key>
        <string>1.0</string>
        <key>CFBundleVersion</key>
        <string>23</string>
        <key>ProjectName</key>
        <string>CPUMonitor</string>
        <key>ReleaseStatus</key>
        <string>GM</string>
        <key>SourceVersion</key>
        <string>23.1</string>
```

```
    </dict>
</dict>
</plist>
```

How Installer Uses Bundle Versions Information

Installer reads the contents of the bundle versions file to determine the versions of the items it is installing. For each `version.plist` path referenced in the bundle versions file, Installer looks for a corresponding path on the user's target volume. If the item exists, Installer reads the `version.plist` file from disk and compares the versions computed from the two `version.plist` files (see "How Installer Computes a Version" (page 109) for details).

If the version on the target volume is newer than the one in the package being installed, Installer determines the path to the owner of the `version.plist`. For example, if Installer encounters a path to a version property list of `./Applications/iTunes.app/Contents/version.plist`, Installer knows that the path to the owner of this property list is `./Applications/iTunes.app`. After determining the owner, Installer prunes any paths beginning with `./Applications/iTunes.app` from the list of paths it will be installing. In this way, Installer can selectively filter out items in the package that would be downgrading items on a user's disk.

If the version number on the target volume is equal to or less than that being installed, or for software for which it doesn't find a version property list, Installer allows all files in the bundle being examined to be installed.

For something like a package that installs only localized resources and not the application itself, it is still appropriate to include the `version.plist` file for the application that owns the resources in the bundle versions file. Installer can then skip installing localized resources if an application on disk is newer than the one described in the bundle versions file of the package.

A shortcoming of this technology is that it currently will not prevent resources from a newer version of a bundle from being installed into an older version of a bundle. For example, it would be possible to install a package containing only Chinese resources for iTunes 5.0 (if such a version existed) into a copy of iTunes 2.0 (or anything less than 5.0).

How Installer Computes a Version

To create a version from a `version.plist` file, Installer computes a 5-tuple from three keys in the `version.plist` file: the `CFBundleShortVersionString` key, the `SourceVersion` key, and the `BuildVersion` key. For a 5-tuple of the form a.b.c.d.e, a.b.c is obtained from the value of `CFBundleShortVersionString`, d is obtained from the value of `SourceVersion`, and e is obtained from the value of `BuildVersion`. If any key is missing, the corresponding element is replaced with 0.

The fourth and fifth elements of the 5-tuple are created from the integer values of `SourceVersion` and `BuildVersion`. If the only difference between two submissions is that the SourceVersion field changed from `1.8.4GMc2` to `1.8.4GMc3`, Installer treats them as the same because the integer values of both of these is 1. For Installer to be able to distinguish between two copies of a bundle, either the `SourceVersion` needs to change for each submission, or the `CFBundleShortVersionString` needs to change, such that the integer values can be differentiated.

Testing With Bundle Versions

The accuracy of the version information as well as the correctness of installation may be judged by manually installing the package with the Installer application, located in `/Applications/Utilities`. Open the installation package and choose File > Show Log. The Installer log can be saved to disk or printed by clicking the Save or Print buttons in the Installer Log window. If there are any version conflicts, the resulting log contains output similar to that shown in Listing 3.

Listing 3 Version info from the Installer log

```
./System/Library/Extensions/AppleMesh.kext (version 1.1.0.1.1) was not installed
 because a newer version (1.1.0.1.60) already exists.
./System/Library/Extensions/IOUSBFamily.kext/Contents/PlugIns/IOUSBLib.bundle
(version 1.8.3.1.1) was not installed because a newer version (1.8.3.1.3) already
 exists.
./System/Library/PrivateFrameworks/DiscRecording.framework (version 1.2.0.1.1)
 was not installed because a newer version (1.2.0.1.30) already exists.
./System/Library/Extensions/AppleStorageDrivers.kext/Contents/PlugIns/LSI-FW-500.kext
 (version 1.1.0.1.2) was not installed because a newer version (1.1.0.1.3)
already exists.
./System/Library/Extensions/System.kext/Contents/PlugIns/IOSystemManagement.kext
 (version 1.1.0.8.1) was not installed because a newer version (1.1.0.8.15)
already exists.
./System/Library/Extensions/System.kext/Contents/PlugIns/IOADBFamily.kext (version
 1.1.0.8.1) was not installed because a newer version (1.1.0.8.15) already
exists.
```

Given the output in the log and knowledge of how Installer computes version 5-tuples, you should be able to track down and correct any problems with the installation.

Specifying Installation Requirements

"Specifying Installation Requirements" describes how to insert information into the information property list for a package to request or require that certain criteria be satisfied before the package is installed.

Important: This requirements mechanism is available starting with Mac OS X version 10.3. It is not currently accessible through the PackageMaker user interface.

You can use the requirements mechanism to ensure that your software is installed only on systems that meet specified criteria, such as minimum hardware or system requirements or the presence of required files or directories. You can also ensure that your software is installed only on volumes that meet specified criteria.

Before reading this article, you should be familiar with the information in "Examining and Modifying Property Lists" (page 87).

New and Existing Requirements Checking

As of Mac OS X version 10.3 (v10.3), Installer supports two approaches to checking installation requirements:

- The requirements mechanism: allows you to specify requirements by inserting information into the information property list for a package.

  Available only in Mac OS X v10.3 and later.

- The InstallationCheck and VolumeCheck executables (typically implemented as shell scripts): allows you to add scripts to a package's `Resources` folder to check for requirements. For details, see "Checking the Installation With InstallationCheck" (page 125) and "Checking the Installation Volume With VolumeCheck" (page 133).

  Available prior to Mac OS X v10.3. Scripts will work in v10.3, but are ignored if the package also includes requirements information in its property list.

The requirements mechanism offers the benefits that it can be easier to use than working with InstallationCheck and VolumeCheck shell scripts and presents fewer security risks. However, for maximum flexibility, you can provide both requirements information and scripts in the same package.

Important: Starting in Mac OS X v10.3, installer applications may present the user with a security warning before an InstallationCheck or VolumeCheck tool is executed.

Installer runs InstallationCheck and VolumeCheck executables, if present, at well-defined times during an installation, as described in "Installation Steps, in Order" (page 54). In Mac OS X v10.3, Installer does similar checking at similar times for a package that contains requirements specifications in its property list:

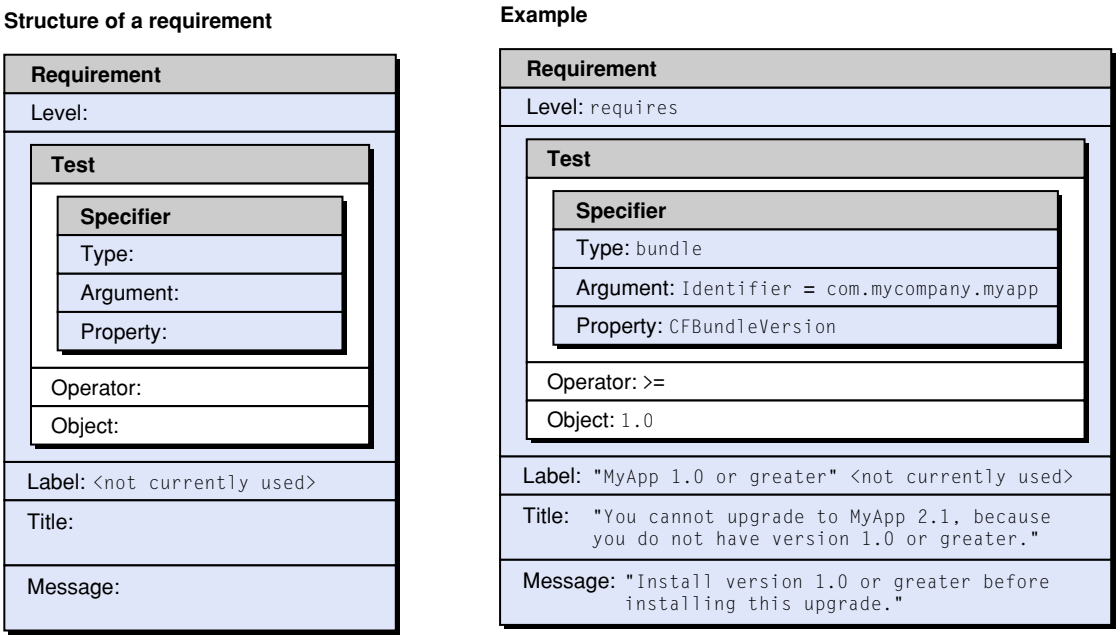
- At the time Installer would execute InstallationCheck, if present, it instead makes a pass through all the requirements, ignoring any requirements that are volume-specific. For example, it will evaluate `gestalt` and `sysctl` requirements, but skip `plist` and `bundle` requirements. (These terms are defined below.)
- At the time Installer would execute VolumeCheck, if present, it instead makes a pass through all the requirements, once for each volume, ignoring any requirements that are *not* volume-specific.
- For a metapackage, requirements are checked first for the metapackage itself, then for any packages it contains that specify requirements. Checking takes place at the same time it would for individual packages.

The Requirements Mechanism

The requirements mechanism is based on the commonly used practice of adding key-value pairs to an information property list—in this case, the property list of an installation package. A **requirement** is a well-defined collection of key-value pairs that together specify criteria that Installer checks before installing the contents of the package. For a complete list of these keys, including information on which are required and which are optional, see "Requirements Keys" (page 117).

The general structure of a requirement is shown on the left in Figure 1. The example on the right shows possible values for a requirement—in this case, a requirement that version 1.0 or greater of the application already be installed:

Figure 1 The installation requirements model



Each requirement contains several components:

- A **level** specifies how the requirement should be enforced (whether it is required, recommended, and so on).

- A **specifier** is a reference to a state or resource in the system upon which software can be dependent. Examples include the version of the iTunes application, the value of a hardware registry key, or the value within a preference plist.
- A **test** is a statement which uses the specifier and resolves to a value of `true` or `false`.
- **Label**, **title**, and **message** provide access to developer-supplied strings that Installer can use to describe the requirement to the user.

A specifier is resolved to an object value, then the object value is compared within the test. For example, a specifier might represent `the version of iTunes` and the test might be whether that object value `is greater than 3`. The result of the test is a boolean value, which indicates whether or not the test was satisfied.

The result of a test can be interpreted according to the requirement level and can be described to the user with title and message strings you supply, as described in "Requirements Message Strings" (page 114).

Listing 1 shows what the requirement example from Figure 1 might look like in a property list. For additional examples, see "Requirements Examples" (page 121).

Listing 1 A requirement for the MyApp installation package

```
<key>IFRequirementDicts</key>
<array>
    <dict>
        <key>Level</key>
        <string>requires</string>
        <key>SpecType</key>
        <string>bundle</string>
        <key>SpecArgument</key>
        <string>/Applications/MyApp</string>
        <key>SpecProperty</key>
        <string>CFBundleVersion</string>
        <key>TestOperator</key>
        <string>&gt;=</string>
        <key>TestObject</key>
        <string>1.0</string>
        <key>LabelKey</key>
        <string>MyAppLabelKey</string>
        <key>TitleKey</key>
        <string>MyAppTitleKey</string>
        <key>MessageKey</key>
        <string>MyAppMessageKey</string>
    </dict>
</array>
```

When to Use Requirements

Although the requirements mechanism is flexible, in general you should avoid specifying broad requirements such as `Mac OS X >= 10.2.5` or `RAM >= 256 MB`. Such statements are useful in marketing or customer support contexts, but they are not well-defined in the machine-level context of software installation and management. Instead, where possible you should use fine-grained requirements such as the examples shown in Table 1.

Table 1 A simple representation of two requirements

Keys	Example 1	Example 2
Type	bundle	bundle
Argument	AppKit framework	Safari application
Property	CFBundleVersion	CFBundleVersion
Operator	ge (greater than or equal)	ge
Object	663.7	74

These examples define requirements that in simple language equate to "the AppKit framework version must be greater than or equal to 663.7" and "the Safari build version must be greater than or equal to 74". For Safari, that might be the equivalent of "Safari must be greater than or equal to version 1.0.2".

Similarly, avoid using marketing versions, such as CFBundleShortVersionString, when specifying requirements. For example, the use of strings such as "1.0 Beta 2" and "0.9" might be useful in user-level marketing or customer support terms, but would not be not well-defined in the machine-level contexts of software installation and management. Instead, you should use CFBundleVersion when specifying requirements (as shown in examples throughout this article).

For information on editing a property list, see "Examining and Modifying Property Lists" (page 87). For information on the keys you use to define requirements in a property list, see "Requirements Keys" (page 117).

Adding Requirements to the Information Property List

The information property list for a package can contain zero or more requirements. To add requirements to the property list, you add the `IFRequirementDicts` key at the root level. The value associated with this key is an array that contains one or more requirements dictionaries. Each dictionary is comprised of a combination of the keys shown in Table 2 (page 117). Several examples of these dictionaries are provided in "Requirements Examples" (page 121).

For information on editing a property list, see "Examining and Modifying Property Lists" (page 87). For information on the keys you use to define requirements in a property list, see "Requirements Keys" (page 117).

Requirements Message Strings

When you define a requirement in a property list, you can specify various message keys. Installer uses these keys to obtain information to display to the user if the requirement is not satisfied. You also supply the information Installer displays, in the form of a strings file based on the keys you defined. The mechanism works like this:

1. For each requirement, you add entries for any of the three keys `LabelKey`, `TitleKey`, or `MessageKey`.

2. For each key you supply a value.

   For example, you might add a `MessageKey` with the value `"SafariRequirementOne"`.

3. You also supply a localizable strings file named `IFRequirement.strings`. A requirement strings file contains messages in the following format:

   ```
   "key string" = "message string";
   ```

   For the `MessageKey` described in the previous step, the strings file entry might look like this:

   ```
   "SafariRequirementOne" = "You must upgrade to Mac OS X v10.3 before installing
    this software.";
   ```

Format and Usage for Message Strings

When a requirement cannot be satisfied, Installer displays information to the user by assembling the message strings you supply according to its predefined usage pattern. If you supply no keys, Installer displays a generic message.

The following sections describe the available message keys, the kinds of strings you should supply for each key, and Installer's usage for assembling the strings into a message.

LabelKey

Important: Through Mac OS X version 10.3, the `LabelKey` key is not used.

The `LabelKey` specifies a string that provides a concise description of the requirement in grammatical object form. Some examples include:

```
"Mac OS X v10.3 or newer"
"at least 128 MB of memory"
"certain languages"
"a newer version of Java"
```

Because the label may be used in various forms, including lists, it should not be a complete sentence nor include ending punctuation.

TitleKey

The `TitleKey` specifies a string that provides a complete explanation of the requirement and why it could not be satisfied. Some examples include:

```
"Safari cannot be installed on this computer, because it requires a new version
 of the WebKit framework."
"Your computer doesn't have a SuperDrive. Do you want to install iDVD anyway?"
```

In the case of a single failed requirement, Installer displays the following:

```
<TitleKey string> <MessageKey string>
```

If the requirement does not define `TitleKey`, Installer constructs a generic one, where `<name>` represents the package title:

```
"<name> cannot be installed on this computer."
```

If the requirement does not define `MessageKey`, no text is displayed for that key.In the case of multiple failed requirements, Installer ignores any `MessageKey` strings (in Mac OS X version 10.3). Instead, it displays the generic title, followed by the titles from each of the requirements. If a requirement does not define a `TitleKey`, no `TitleKey` string is displayed for it.

```
<name> cannot be installed on this computer.
    <TitleKey1 string>
    <TitleKey2 string>
    <TitleKey3 string>
```

MessageKey

The `MessageKey` specifies a string that provides a short explanation of how to resolve the issue, if possible, when the requirement is not satisfied. Some examples include:

```
"You must upgrade to Mac OS X v10.3 before installing this software."
"Use Software Update to check for a newer version of this driver."
"This volume does not meet the requirements for this system update."
```

`MessageKey` is used in conjunction with `TitleKey`, as described in "TitleKey" (page 115).

Localizing the Requirements Strings File

If you localize your requirements strings file, Installer will display messages based on the user's current Languages preference. You can use any of the folder names listed in "Localized Folder Names" (page 70). These folders should be located within the package's `Resources` directory, so that the resulting location looks something like `Resources/<language>.lproj/IFRequirement.strings`. You can view the hierarchy of a package `Resources` directory containing localized folders in Listing 2 (page 130). You would place your `IFRequirement.strings` file in place of the `InstallationCheck.strings` file in that listing.

If no `IFRequirement.strings` file is placed in a localized directory, or no localized version is found for the user's current language selection, Installer looks for a default version at the highest level in the `Resources` directory.

Note: Installer relies on the bundle-searching services provided by CFBundle, and described in "Searching for Bundle Resources" in *Bundle Programming Guide* in Core Foundation Resource Management.

Default Messages Provided by Installer

As noted in "TitleKey" (page 115), if you do not supply message keys or a `IFRequirement.strings` file, Installer supplies a generic message when a requirement is not satisfied.

Requirements Keys

Table 2 lists the keys you use to define a requirement in a package's information property list. These keys correspond to the structure shown in Figure 1 (page 112). Additional information for each key is provided in the sections that follow the table. For information on working with these keys, see "Adding Requirements to the Information Property List" (page 114) and "Requirements Examples" (page 121).

Important: The values for the `LabelKey`, `MessageKey`, and `TitleKey` keys are not themselves strings to be displayed. Rather, they are treated as keys that identify strings within a localizable strings file you supply (described in "Requirements Message Strings" (page 114)).

Table 2 Keys used within a requirements dictionary key

Key	Type	Required	Summary
`LabelKey`	String	No	Key to a short description of the requirement in a localizable strings file. Not currently used.
`TitleKey`	String	No	Key to a description in a localizable strings file that provides a complete explanation of the requirement and why it could not be satisfied.
`MessageKey`	String	No	Key to a short description in a localizable strings file of how to address the problem if the requirement is not satisfied.
`Level`	String	No	Specifies the degree of enforcement. Default is `requires`.
`SpecArgument`	any	Yes	The argument part of the specifier for the requirement test. Specific to the `SpecType`.
`SpecProperty`	any	No	Part of the specifier for the requirement test. Specific to the `SpecType`.
`SpecType`	String	Yes	The type of the specifier. Evaluated according to `SpecArgument` and `SpecProperty` to resolve the specifier.
`TestObject`	any	No	The right-hand object value. By default, you don't need to supply an entry for this key, unless otherwise stated.
`TestOperator`	String	No	Operator used to test the left-hand value (or specifier) against the right-hand value (or object). The default value is =.

LabelKey, TitleKey, and MessageKey

The value for a `LabelKey`, `TitleKey`, or `MessageKey` is a key that specifies a string in a localized strings file named `IFRequirement.strings`, stored in the package's `Resources` folder. These keys and strings files are described in "Requirements Message Strings" (page 114).

All of these keys are optional and if you do not supply them, Installer supplies a generic message when a requirement is not satisfied.

Level

The value supplied for the `Level` key specifies the resulting behavior when a requirement is not satisfied on the user's system at pre-installation time. The currently supported values are:

- `requires`: Used if the package should not be installed on systems where the requirement is not satisfied.

  This is the default value for the `Level` key. It constitutes a technical requirement (as opposed to a legal, marketing, or other non-technical requirement).

  Depending on the requirement strings supplied in the package, Installer might display a message like the following if the requirement is not satisfied:

  "Safari 1.0 requires Java version 1.3 or higher."

- `recommends`: Used to specify anything above and beyond a minimal technical requirement. The package should be installed, even on systems where the recommendation is not satisfied; however, the user experience may be enhanced if the recommendation is satisfied. Some clients, including Installer, may display a recommendation to the user, such as the following:

  "iTunes 4 recommends QuickTime 6.2 or higher for full functionality."

SpecArgument, SpecProperty, and SpecType

Together, the `SpecType`, `SpecArgument`, and `SpecProperty` keys constitute an object specifier for a particular system resource or state. Installer resolves the object specifier to return the actual object value for the user's system, if possible. Examples of object specifiers include the version number of a given application or package receipt, a `gestalt` or `sysctl` value, or a specific value within a property list file or hardware registry. See "Requirements Examples" (page 121) for samples of how to combine the type, argument, and property in a requirement.

Table 3 lists specifier types and the arguments and properties you use with them. You can think of the information from the first two columns (type and argument) as specifying a thing, and the third column (property) identifying some attribute of that thing. In some cases, you supply just the first two items to specify a requirement, while in other cases all three are required.

Additional information for each specifier type is provided in the sections that follow the table.

Table 3 Specifier types

SpecType	SpecArgument	SpecProperty	Specifies
`package`	Package identifier	Property list key	A property list value from an Installer package (with extension `.pkg`)
`sysctl`	a `sysctl` name	none	A `sysctl` entry. Can specify various system information
`ioregistry`	IORegistry path	Key path	An IOKit Registry object (which identifies a device)
`gestalt`	Gestalt selector	none	A Carbon Gestalt selection

SpecType	SpecArgument	SpecProperty	Specifies
`bundle`	a standard or tokenized file path	Property list key (to test for existence, none)	A property list value from the specified bundle
`plist`	a standard or tokenized file path	Key path (to test for existence, none)	A possibly nested property list value from the specified property list
`file`	a standard or tokenized file path	NSFileManager attribute key (to test for existence, none)	Various information about a flat file

Package

Specifies an Installer package. The `SpecArgument` value should be a package identifier string (currently a bundle identifier string from the package's info property list). You use this property to set a requirement within a metapackage that another package be present. You currently can only use this type of requirement in a metapackage. You can't use this property to look up installed receipts on disk—it will access only "full" packages.

The `SpecProperty` value should be a property list key string in the package. You should always supply a `SpecProperty` entry.

Sysctl

Specifies a `sysctl` entry. The `SpecArgument` value should be a string specifying a MIB style `sysctl` name. See the man page for `sysctl(8)` for values. You can specify many system attributes, including the amount of physical memory (`hw.sysmem`) or the hardware model (`hw.model`).

An example of a value for a `sysctl` entry is `hw.ncpu`, which would specify a requirement based on the number of CPUs.

You don't supply a `SpecProperty` entry for `sysctl`.

IORegistry

Specifies an IOKit Registry object. The `SpecArgument` value should be a string specifying an IORegistry path. The format is `plane:path`. For example:

```
IOService:/MacRISC2PE/pci@f0000000/AppleMacRiscAGP/*/*/*/AppleBacklightDisplay
```

The `SpecType` and `SpecArgument` together specify a hardware device in the IORegistry. The `SpecProperty` further refines the requirement by specifying an attribute of the device. The property should be a string that specifies a key path. Wildcards are supported using the '*' character for nodes in the registry path, as well as for nodes in the key path.

The following example specifies the brightness value of the device:

```
IODisplayParameters:brightness:value
```

For more information on IOKit, see *I/O Kit Fundamentals* and other documents in Darwin Documentation.

Gestalt

Specifies a Carbon Gestalt selector. You can use Gestalt selectors to identify many attributes of the operating environment. The `SpecArgument` value should be a four character code string specifying a Gestalt selector.

You don't supply a `SpecProperty` entry for `gestalt`.

For information on Gestalt, see *Gestalt Manager Reference* in Carbon Resource Management Documentation.

Bundle

Specifies a bundle (application, framework, plug-in, and so on). The `SpecArgument` value is a standard or tokenized file path. A standard path is just a path, such as `/Applications/MyApplication.app`. Tokenized paths are described in "Token Definitions" (page 97).

The `SpecProperty` value should be a string value from the property list of the specified bundle. If you just want to check whether the bundle [or file] exists, don't supply a `SpecProperty` entry. In that case, the requirement will be satisfied if the bundle exists and not satisfied if it doesn't.

Plist

Specifies a property list file. The `SpecArgument` value is a standard or tokenized file path to the specified file.

The `SpecProperty` value should be string specifying a key path. That is, for a top-level key, the property could be one word, such as `NSAppleScriptEnabled` (a key which indicates a Cocoa application uses Cocoa's built-in AppleScript support). A nested property might have multiple keys, joined by periods, such as `PhonePrefs.RingerPrefs.CurrentTune`. Wildcards are supported using the ' *' character. Integers are used as array indices (that is, an index into the array for an array key).

File

Specifies a flat file. The `SpecArgument` value is a standard or tokenized file path. A standard path is just a path to a file. Tokenized paths are described in "Token Definitions" (page 97).

The `SpecProperty` value should be a file attribute constant string, defined in NSFileManager. There are many file attributes you can incorporate in a requirement, such as file size (NSFileSize) or creation date (NSFileCreationDate).

If you just want to check whether the file exists, don't supply a `SpecProperty` entry. In that case, the requirement will be satisfied if the file exists and not satisfied if it doesn't.

For example, if your package requires the presence of the file `/mach_kernel`, you can specify a `SpecType` of `file`, a `SpecArgument` of `/mach_kernel`, and no `SpecProperty`. As a result, Installer will only allow the package to be installed if the file `/mach_kernel` exists.

TestObject

The TestObject key specifies the "required" object value. See "TestOperator" (page 121) for an explanation of how it is used.

TestOperator

The `TestOperator` key is used as follows: the left value (specified by the keys described in "SpecArgument, SpecProperty, and SpecType" (page 118)) is tested with the operator specified by the `TestOperator` key against the right value specified by the `TestObject` key. As a value for the `TestOperator` key, you can use any of the comparison operators shown in the two right-hand columns in Table 4 (XML notation and Perl notation).

Important: Because the property list of a package is in XML format, you must escape special characters, such as "<" and ">". Examples are provided below.

Table 4 Comparison operators for the TestOperator key

Operator	Symbol	XML notation	Perl notation
equal	=	=	eq
less than or equal	<=	<=	le
less than	<	<	lt
greater than or equal	>=	>=	ge
greater than	>	>	gt
not equal	!=	!=	ne

For example, the full value entry for greater than or equal in XML notation is:

```
<string>&gt;=</string>
```

The full value entry for greater than or equal using Perl notation is:

```
<string>ge</string>
```

Requirements Examples

This section provides examples of various common requirements definitions.

Bundle Requirements

Listing 2 shows a requirements definition that specifies a bundle version. This type of requirement is useful for software that links against external frameworks. In this example, the package requires version 663.7 or greater of the AppKit framework.

Listing 2 A requirements definition for a minimum framework version

```
<key>IFRequirementDicts</key>
```

```
<array>
    <dict>
        <key>SpecType</key>
        <string>bundle</string>
        <key>SpecArgument</key>
        <string>/System/Library/Frameworks/AppKit.framework</string>
        <key>SpecProperty</key>
        <string>CFBundleVersion</string>
        <key>TestOperator</key>
        <string>&gt;=</string>
        <key>TestObject</key>
        <string>663.7</string>
    </dict>
</array>
```

See "TestOperator" (page 121) for a description of the valid test operators (such as `&gt:=` in this example).

If you're packaging a relocatable application, you can use the same format shown in Listing 2, but also provide a token definition to specify that Installer should search for the item on disk and upgrade older versions as appropriate. You need to do this only if your package must upgrade or otherwise overwrite the relocated files. For a complete description of searching and token definitions, see "Finding Previously Installed Software" (page 97).

Property List Requirements

Listing 3 shows a requirements definition that specifies a simple test based on the bundle version value from a property list. In this example, the package requires Mac OS X version 10.3 or greater.

Listing 3 A requirements definition for a value from a property list

```
<key>IFRequirementDicts</key>
<array>
    <dict>
        <key>SpecType</key>
        <string>plist</string>
        <key>SpecArgument</key>
        <string>/System/Library/CoreServices/SystemVersion.plist</string>
        <key>SpecProperty</key>
        <string>ProductVersion</string>
        <key>TestOperator</key>
        <string>&gt;=</string>
        <key>TestObject</key>
        <string>10.3</string>
    </dict>
</array>
```

File Requirements

Listing 4 shows a requirements definition that specifies a flat file requirement. In this example, the package is not installed if the specified font file is present.

Listing 4 A requirements definition for a file

```
<key>IFRequirementDicts</key>
<array>
    <dict>
        <key>SpecType</key>
        <string>file</string>
        <key>SpecArgument</key>
        <string>/Library/Fonts/Apple Chancery.dfont</string>
        <key>TestOperator</key>
        <string>eq</string>
        <key>TestObject</key>
        <string></string>
        <key>Level</key>
        <string>requires</string>
        <key>LabelKey</key>
        <string>AppleChanceryLabel</string>
        <key>MessageKey</key>
        <string>AppleChanceryMessage</string>
    </dict>
</array>
```

The requirement effectively states that the specifier must resolve to an empty string. This evaluates to `true` if the file doesn't exist in the Fonts folder on the installation volume, `false` if it does exist.

Checking the Installation With InstallationCheck

You can prevent users from installing your software on systems that do not meet certain criteria, such as minimum hardware requirements or the presence of required files or directories. To do so, you supply an executable file, named `InstallationCheck`, in your installation package that checks the criteria you are interested in. If those criteria are not met, you can cancel installation and supply a meaningful message to the user. When you cancel installation in this way, Installer exits and the installation cannot be continued.

Important: Starting in Mac OS X version 10.3, installer applications may present the user with a security warning before a InstallationCheck tool is executed.

For another possible approach to specifying requirements, see "Specifying Installation Requirements" (page 111).

InstallationCheck Overview

The Installer application looks for the InstallationCheck executable file in the `Resources` directory of the package. If found, the file is executed after authentication (if required) but before any other operations. See "The Installation Process" (page 53) for a look at the larger picture of what happens during an installation.

Important: The InstallationCheck tool or script must have its executable bit set. This is done automatically when the package is built with PackageMaker.

When a metapackage is opened, Installer runs the InstallationCheck script of any package that includes one. However, it does not run an InstallationCheck script that is present in the metapackage itself.

InstallationCheck can use any criteria to allow or disallow installation. InstallationCheck returns a value that tells Installer whether or not to continue with installation. For details on this value, see "Return Value for the InstallationCheck Executable" (page 127).

Note: Most executable files used for installation checking are likely to be shell scripts. Such scripts can use any shell, as long as it starts with a standard `#!` comment to identify the shell.

When an installation is canceled, the value returned by InstallationCheck also indicates what message Installer displays. Installer obtains the message from an `InstallationCheck.strings` file you supply in the installation package. If you supply strings files for multiple locales, Installer displays the appropriate version based on the user's current Languages preference. If you do not supply a message, Installer supplies a default message.

An InstallationCheck script is not the correct place to try to "fix" issues such as incorrect file permissions (for more on permissions, see "Authorization, File Ownership, and Permissions" (page 79)). In general, you should perform installation checking only when absolutely necessary.

The following sections describe

- where the `InstallationCheck` file and message files are placed in the installation package
- the arguments and environment variables available to InstallationCheck
- the return values InstallationCheck uses to specify an action and a message
- the default messages supplied by Installer
- the format of messages in a message file
- a sample `InstallationCheck` script file

In addition, see "Working With Script Files" (page 147) for tips on using executable files with PackageMaker and Installer.

File Placement for Installation Checking

Each file executed as part of an installation, including `InstallationCheck`, should be placed in the `Resources` directory of the package (or metapackage) and must have its execution bit set. In the example described in "Creating a Package" (page 43), you place script files in an `Install_resources` directory, then choose that directory as your `Resources` directory in PackageMaker.

If you place the file correctly and build the package with PackageMaker, it will automatically set the execution bit for the file. However, you may need to set explicit permissions for the executable—for more information, see "Authorization, File Ownership, and Permissions" (page 79).

Any `InstallationCheck.strings` message files you provide should also be placed in the `Install_resources` directory (and end up in the package's `Resources` directory), within a localized resource directory with the appropriate name. For example, the items labeled `-1-` through `-5-` in Listing 1 show the final file placement in a test package, whose strings files are localized for English and French. For information on which localized folder names you can use, see "Localized Folder Names" (page 70).

Listing 1 Location of InstallationCheck executable and strings files in a package

```
Test.pkg
    Contents
        Archive.bom
        Archive.pax.gz
        Info.plist
        Resources
            Description.plist
            InstallationCheck                -1-
            English.lproj                    -2-
                InstallationCheck.strings    -3-
            French.lproj                     -4-
                InstallationCheck.strings    -5-
```

Warning: If no `InstallationCheck.strings` files are placed in localized directories, or no localized version is found for the user's current language selection, Installer should look for a default version at the highest level in the `Resources` directory. Due to a bug, however, only string files in localized resource directories are currently used.

Arguments and Environment Variables

When Installer executes the `InstallationCheck` file, it provides four arguments:

- `$1`: The full path to the installation package. For example:

  `/Volumes/Projects/Testing/Simple_Carbon_App.pkg`

- `$2`: The full path to the installation destination. For example:

  `/Applications`

- `$3`: The mountpoint of the destination volume. For example:

  `/` or `/Volumes/External_Drive`

- `$4`: The root directory for the current System folder:

  `/`

Warning: The values for the variables `$2`, `$3`, and `$4` are always set to `/`, and should not be counted on in your `InstallationCheck` script (even in the situation where you might expect that value).

Installer also makes one environment variable available:

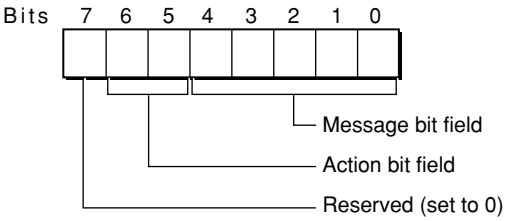
- `$SCRIPT_NAME`: The name of the file being executed. That is:

  `InstallationCheck`

Return Value for the InstallationCheck Executable

InstallationCheck returns a value that Installer interprets as a pair of bit fields, as shown in Figure 1. Bits 5 and 6 of the return value determine the action—that is, whether to continue the installation or to cancel it (with a message of explanation). When an installation is cancelled, Installer exits and the installation cannot be continued.

Figure 1 Bit fields in the value returned by InstallationCheck



The bit field consisting of bits 0–4 specifies a message index. Depending on the value of the action bits (bits 5 and 6), Installer uses the index to obtain a message string from the `InstallationCheck.strings` file. The strings file currently must be in a localized directory, as described in "File Placement for Installation Checking" (page 126). If no strings file is provided or if the specified message cannot be found, Installer provides a default message.

Important: Bit 7 is reserved for future use and must be 0. As a general rule, set all unused bits to 0. You can do this by setting the return value to 0, then shifting in the required bits, as shown in Table 1.

Table 1 describes how Installer is intended to interpret the action bits (bits 5 and 6) in the value returned by InstallationCheck, and also notes a couple of places where, due to a bug, your results may vary from the intended results.

Table 1 How Installer should respond to value in bits 5 and 6

Bit values (6 5)	Numerical value	Obtain by shifting	Action
0 0	0	(not needed)	no error; continue installation
0 1	32	1 << 5	**intended result:** displays a warning message, but allows installation to continue
			possible bug: allows installation to continue, but may show no error message
1 0	64	1 << 6	**intended result:** cancels the installation without any message (fails silently)
			possible bug: cancels the installation, but may display message "This software cannot be installed on this computer"
1 1	96	3 << 5	displays a warning message to the user, then cancels the installation

Table 2 shows possible bit field values for the message bit field (bits 0 through 4) in the value returned by InstallationCheck. The five bits allow for a value between 0 and 31. Table 3 (page 129) shows default string values. Listing 2 (page 130) shows a sample strings file.

Table 2 Values for bits 0 through 4 and resulting messages

Value	Message
0	Should not be 0 unless the action bit is also 0 (no error)
1–4	Specifies a default string provided by Installer, as shown in Table 3 (currently only used when action bits have value 96; that is, both action bits are 1)
5–15	Reserved for future use by Installer
16–31	Specifies a message string in an `InstallationCheck.strings` file in the package, as shown in "Strings Files for the InstallationCheck Executable" (page 130) (currently only used when action bits have value 96; that is, both action bits are 1)

If the package contains strings files for multiple locales, Installer displays the appropriate version based on the user's current Languages preference. For a sample strings file, see "Strings Files for the InstallationCheck Executable" (page 130).

InstallationCheck Message Strings

As described in "Return Value for the InstallationCheck Executable" (page 127), Installer either continues the installation (without warning) or cancels it (with a message of explanation), depending on the return value from InstallationCheck.

Default Messages Provided by Installer

When the action bits are `11` (or value 96), Installer provides default message strings based on the value InstallationCheck returns in the message bit field (bits 0 to 4), as described in Table 3.

Table 3 Installer default messages for InstallationCheck

Value	Message string
1	This software cannot be installed on this computer.
2	The software <package name> cannot be installed on this computer.
3	<package name> cannot be installed on this computer.
4	An error was encountered while running the InstallationCheck tool for package <package name>.
5–15	reserved for future use by Installer

Strings Files for the InstallationCheck Executable

To display a message when cancelling an installation, you provide one or more files named `InstallationCheck.strings` and place them in the appropriate localized directory within the package's `Resources` directory. "File Placement for Installation Checking" (page 126) describes the placement of strings files in the package.

A strings file contains messages in the following format:

```
"number string" = "message string";
```

where `"number string"` represents a numeric value between 16 and 31, as described in Table 2 (page 129) and `"message string"` is the string to display. Strings always end with a semicolon. An English strings file with two messages might look like the one in Listing 2.

Listing 2 A sample InstallationCheck.strings file

```
"16" = "Can't install unless you have a Super Drive.";
"17" = "Can't install unless you have more than 128MB of RAM.";
```

Installer only displays strings from the strings file if the action bits returned by InstallationCheck are `11` (or a value of 96). So to specify an index in your strings file, you add a value between 16 and 31 to 96. For example, if the message index returned to the Installer is 112 (or a bit value of `1101000`, indicating the installation should not be allowed and the message index is 16), the message displayed to the user is

```
Can't install unless you have a Super Drive.
```

The bit fields in the value returned by InstallationCheck are described in detail in "Return Value for the InstallationCheck Executable" (page 127).

InstallationCheck Example

Listing 3 shows a template for an InstallationCheck script. This script just displays its arguments and environment variables, then returns a value that causes Installer to stop the installation and display an error message. However, you can plug your installation check testing into this script.

You can examine output from InstallationCheck in the Console application (available in `/Applications/Utilities`). Alternatively, you can redirect output to a log file, using script statements like the following:

```
echo "Full volume path is: $path" >> /tmp/scripts.log
```

The first line in this script (`#!/bin/bash`) indicates that the script uses the `bash` shell, located in `/bin`. Other lines that begin with the `#` character are comments.

Note: Although this script has been tested, use care in cutting and pasting it from this document. For more information, see "Troubleshooting Packages and Installation" (page 145).

Listing 3 An InstallationCheck script that cancels installation and displays a message from the strings file

```
#!/bin/bash
#
# This installation script just echoes the values of the available
# arguments and environmental variables.
#
echo "START INSTALLATIONCHECK SCRIPT"
echo ""
echo "Arguments:"
echo ""
echo "\$1: the full path to the install package."
echo "     $1"
echo "\$2: the full path to the install destination."
echo "     $2"
echo "\$3: the mountpoint of the destination volume."
echo "     $3"
echo "\$4: the root directory \"/\" for the current System folder."
echo "     $4"
echo ""
echo "Environment variables:"
echo ""
echo ""
echo "\$SCRIPT_NAME: $SCRIPT_NAME"
echo ""
let retval=112
echo "\$retval = $retval"
echo "END INSTALLATIONCHECK SCRIPT"
exit $retval
```

Checking the Installation Volume With VolumeCheck

You can prevent users from installing your software on volumes that do not meet certain criteria. To do so, you supply an executable file, named `VolumeCheck`, in your installation package that determines whether a volume should be allowed as a possible installation destination.

Important: Starting in Mac OS X version 10.3, installer applications may present the user with a security warning before a VolumeCheck tool is executed.

For another possible approach to specifying volume requirements, see "Specifying Installation Requirements" (page 111).

VolumeCheck Overview

The Installer application looks for the VolumeCheck executable file in the `Resources` directory of the package. If found, the file is executed before Installer asks the user to choose an installation volume (and possibly a specific directory). This takes place after the user has already performed any required preliminary steps, such as authenticating or accepting a license agreement. See "The Installation Process" (page 53) for a look at the larger picture of what happens during an installation.

Important: The VolumeCheck tool or script must have its executable bit set. This is done automatically when the package is built with PackageMaker.

Note: Most executable files used for volume checking are likely to be shell scripts. Such scripts can use any shell, as long as it starts with a standard `#!` comment to identify the shell.

Installer runs VolumeCheck once for each currently mounted volume. VolumeCheck can use any criteria to accept or reject a volume. VolumeCheck returns a value that tells Installer whether or not to allow the volume as an installation destination. For details on this value, see "Return Value for the VolumeCheck Executable" (page 135). If a volume is disallowed, Installer displays its icon with a badge that indicates the volume can't be used for this installation. If the user attempts to select the volume anyway, you can supply a message that explains why that volume is not suitable.

When VolumeCheck returns a value that indicates that a volume should be disallowed, the return value also indicates what message to display if the user clicks the icon for that volume. Installer obtains the message from a `VolumeCheck.strings` file you supply in the installation package. If you supply `VolumeCheck.strings` files for multiple locales, Installer displays the appropriate version based on the user's current Languages preference. If you do not supply a message, Installer supplies a default message.

The following sections describe

- where the `VolumeCheck` file and message files are placed in the installation package

- the arguments and environment variables available to VolumeCheck
- the return values VolumeCheck uses to specify an action and a message
- the default messages supplied by Installer
- the format of messages in a message file
- a sample `VolumeCheck` script file

In addition, see "Working With Script Files" (page 147) for tips on using executables with PackageMaker and Installer.

File Placement for Volume Checking

Each file executed as part of an installation, including `VolumeCheck`, should be placed in the `Resources` directory of the package and must have its execution bit set. If you place the file correctly and build the package with PackageMaker, it will automatically set the execution bit for the file. However, you may need to set explicit permissions for the executable file—for more information, see "Authorization, File Ownership, and Permissions" (page 79).

Any `VolumeCheck.strings` message files you provide should also be placed in the `Resources` directory, within a localized resource directory with the appropriate name. For example, the items labeled `-1-` through `-5-` in Listing 1 show the file placement in a test package, where the strings files are localized for English and French. For information on which localized folder names you can use, see "Localized Folder Names" (page 70).

Warning: If no `VolumeCheck.strings` files are placed in localized directories or no localized version is found for the user's current language selection, Installer should look for a default version at the highest level in the `Resources` directory. Due to a bug, however, only string files in localized resource directories are currently used.

Listing 1 Location of VolumeCheck executable and strings files in a package

```
Test.pkg
    Contents
        Archive.bom
        Archive.pax.gz
        Info.plist
        Resources
            Description.plist
            VolumeCheck              -1-
            English.lproj            -2-
                VolumeCheck.strings  -3-
            French.lproj             -4-
                VolumeCheck.strings  -5-
```

Arguments and Environment Variables

When Installer executes the `VolumeCheck` file, it provides one argument:

- `$1`: The full path to the volume to evaluate. For example:

  `/Volumes/Mac_OS_X_Work`

Installer also makes one environment variable available:

- `$PACKAGE_PATH`: The full path to the installation package. For example:

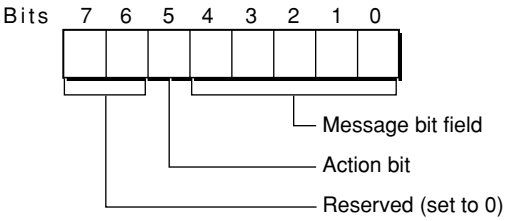
  `/Volumes/Projects/Testing/Simple_Carbon_App.pkg`

  VolumeCheck can use this path to obtain resources within the package.

Return Value for the VolumeCheck Executable

VolumeCheck returns a value that Installer interprets as a pair of bit fields, as shown in Figure 1. Bit 5 of the return value determines the action—that is, whether to allow or disallow the volume as a destination for the installation.

Figure 1 Bit fields in the value returned by VolumeCheck



If a volume is disallowed, Installer displays the icon for that volume with a badge that indicates it can't be used for this installation. It also saves the value from the second bit field, consisting of bits 0–4. If the user later clicks the icon for the disallowed volume, Installer displays a message in the following format:

```
You cannot install this software on this disk. <message>
```

Installer uses the value from the second bit field to obtain the string `<message>` from the `VolumeCheck.strings` file. The strings file currently must be in a localized directory, as described in "File Placement for Volume Checking" (page 134). If no strings file is provided or if the specified message cannot be found, Installer provides a default message. Possible bit field values are shown in Table 2. Default string values are shown in Table 3 (page 136). A sample strings file is shown in Listing 2 (page 137).

Important: Bits 6 and 7 are reserved for future use and must be zero. As a general rule, set all unused bits to 0.

Table 1 describes how Installer interprets the action bit (bit 5) in the value returned by VolumeCheck:

Table 1 Actions specified by value in bit 5

Bit value	Action
0	Allow installation on the volume

Bit value	Action
1	Disallow installation and add a badge to the icon for the volume

Table 2 describes how Installer interprets the message bits (bits 0 through 4) in the value returned by VolumeCheck. The five bits allow for a value between 0 and 31.

Table 2 Values for bits 0 through 4 and resulting messages

Numeric value	Message
0	Should not be 0 unless the action bit is also 0
1–2	Specifies a default string provided by Installer, as shown in Table 3
3–15	Reserved for future use by Installer
16–31	Specifies a message string in a `VolumeCheck.strings` file in the package, as shown in "Strings Files for the VolumeCheck Executable" (page 137); if the value does not have a matching message string, Installer displays the value itself to the user; for example: You cannot install this software on this disk. 31

If the package contains strings files for multiple locales, Installer displays the appropriate version based on the user's current Languages preference. For a sample strings file, see "Strings Files for the VolumeCheck Executable" (page 137).

VolumeCheck Message Strings

As described in "Return Value for the VolumeCheck Executable" (page 135), if a user clicks an icon for a disallowed volume, Installer displays a message based on a bit-field value specified previously by VolumeCheck. The message begins with "You cannot install this software on this disk." It is followed by the specified message string from the file `VolumeCheck.strings`, which may be stored in a localized resource directory in the installation package. If the package does not supply a strings file or if the file doesn't contain the specified message, Installer displays a default message.

Default Messages Provided by Installer

Based on the value VolumeCheck returns in the message bit field (bits 0–4), Installer provides the default message strings described in Table 3. These messages are concatenated with the string "You cannot install this software on this disk. "

Table 3 Installer default messages for VolumeCheck

Numeric value	Message string
0	You are not allowed to install the software on this disk for an unknown reason.

Numeric value	Message string
1	Could not find specified message for index 1.
2	An error was encountered while running the VolumeCheck tool for package <package name>.
3–15	Could not find specified message for index <value>

Strings Files for the VolumeCheck Executable

To display a message when a user clicks the icon for a disallowed volume, you provide one or more files named `VolumeCheck.strings` and place them in the appropriate localized directory within the package's `Resources` directory. "File Placement for Volume Checking" (page 134) describes the placement of strings files in the package.

A strings file contains messages in the following format:

```
"number string" = "message string";
```

where `"number string"` represents a numeric value between 16 and 31, as described in Table 2 (page 136) and `"message string"` is the string to display. Strings always end with a semicolon. An English strings file with two messages might look like the one in Listing 2.

Listing 2 A sample VolumeCheck.strings file

```
"16" = "Can't install on volumes named Jaguar.";
"17" = "Can't install on volumes named Puma.";
```

For example, if the message index returned to the Installer is 48 (or a bit value of `110000`, indicating the volume should be disallowed and the message index is 16), the message displayed to the user is

```
You cannot install this software on this disk. Can't install on volumes named
Jaguar.
```

The bit fields in the VolumeCheck return value are described in detail in "Return Value for the VolumeCheck Executable" (page 135)

VolumeCheck Example

Listing 3 shows a simple VolumeCheck script. This script checks the volume name and disallows any volumes named `Jaguar`, but a VolumeCheck script is free to perform checking of arbitrary complexity. You can examine output from VolumeCheck in the Console application (available in `/Applications/Utilities`). Alternatively, you can redirect output to a log file, using script statements like the following: `echo "Full volume path is: $path" >> /tmp/scripts.log`

Note: Although this script has been tested, you should use care in cutting and pasting it from this document. For more information, see "Troubleshooting Packages and Installation" (page 145).

The first line in this script (`#!/bin/bash`) indicates that the script uses the `bash` shell, located in `/bin`. Other lines that begin with the `#` character are comments.

Listing 3 A VolumeCheck script that checks the volume name

```
#!/bin/bash
#
# Argument 1 (the only argument to VolumeCheck) is the full path to a volume
#   in Installer's Target Select panel that a user can choose to
#   install the software on.
# The path is of the form "/" or /Volume/{volume name}
#
# For convenience, this script echoes the path and name of each volume.
path=$1
echo "Full volume path is: $path"
#
# Get the volume name from the full path.
#
path=${path##/*/}
echo "Volume name is: $path"
#
# You can perform testing of arbitrary complexity here.
# This script just disallows installing on volumes named "Jaguar".
#
# Note: If $path were not enclosed in quotes in the next statement,
#       the script would get an error if the volume name included a space.
#
if ([ "$path" = "Jaguar" ]) then
#
# If the volume has the disallowed name, return a value that tells Installer
# not to allow installation on the volume.
# The value should also specify a string from the VolumeCheck.strings file
# to display if the user clicks the volume icon.
#
# To do this, supply an error return value of 32.
#
# Sets the action bit (bit 5) to 1, to disallow the volume.
let retval=(1<<5)
#
# Sets the message value bits (bits 0 to 4) to 16,
# specifying the message associated with "16" in the strings file.
#
let retval=(retval|16)
#
else
#
# The volume name is OK, so allow it as a possible destination.
#
retval=0
fi
exit $retval
```

Modifying an Installation With Scripts

You can use scripts (or other executable files) to configure the destination environment before installation or to perform additional processing after installation.

Modifying With Scripts Overview

The Installer application looks for certain named executable files in the installation package and, if found, executes them at specified times during installation. These times take place after the user has performed any required introductory steps, such as authenticating, accepting a license agreement, specifying a volume, and so on. For a detailed description of what takes place during an installation, see "The Installation Process" (page 53).

Important: For pre- and postprocessing to work correctly, each executable tool or script must have its executable bit set. This is done automatically when you build a package with PackageMaker.

Note: Most executable files used for pre-and postprocessing are likely to be shell scripts. Such scripts can use any shell, as long as it starts with a standard #! comment to identify the shell.

The following sections describe:

- the required names for executable files
- when executable files are executed during installation
- where executable files are placed in the installation package
- the arguments passed to an executable
- the environment variables available to an executable

Scripts are not the correct place to try to "fix" issues such as incorrect file permissions (for more on permissions, see "Authorization, File Ownership, and Permissions" (page 79)). In general, the scripts described here should be used only when absolutely necessary.

For tips on working with scripts, see "Working With Script Files" (page 147)

Executable Names

When a user clicks the Install button to start the installation process, Installer looks for the following files at specific points in the installation process and executes them if found. The files must be executable and have the names shown here:

- `preflight`, `postflight`: The preflight file is executed before the installation process and the postflight file after it.
- `preinstall`, `postinstall`: The preinstall file is executed before installing a single package and the postinstall file after installing it. These files are only executed for an install (the software is installed for the first time).
- `preupgrade`, `postupgrade`: The preupgrade file is executed before upgrading a single package and the postupgrade file after upgrading it. These files are only executed for an upgrade (an existing version of the software is replaced, in full or in part).

When Files Are Executed

See "The Installation Process" (page 53) for a look at the larger picture of what happens during an installation.

Execution Order for a Single Package

For a single package installation, Installer performs the following steps:

1. Executes the `preflight` file (if present).
2. For an install, executes the `preinstall` file (if present).

   For an upgrade, executes the `preupgrade` file (if present).

   For information on how Installer determines whether to perform an install or an upgrade, see "Installs and Upgrades" (page 33).
3. Extracts the package's files and places them on the target volume.
4. For an install, executes the `postinstall` file (if present).

   For an upgrade, executes the `postupgrade` file (if present).
5. Executes the `postflight` file (if present).

Execution Order for a Metapackage

See "Execution Order for a Complex Metapackage" (page 56) for information on which files are executed and when.

Executable File Placement in a Package

All files executed as part of an installation should be placed in the `Resources` directory of the package. When you build a package with PackageMaker, this is done automatically. The items labeled `-1-` through `-6-` in Listing 1 show the placement in a test package.

Listing 1 Location of executable files in a package

```
Test.pkg
    Contents
        Archive.bom
        Archive.pax.gz
        Info.plist
        PkgInfo
        Resources
            Description.plist
            postflight      -1-
            postinstall     -2-
            postupgrade     -3-
            preflight       -4-
            preinstall      -5-
            preupgrade      -6-
```

Arguments

When Installer runs an executable file, it provides the following arguments:

- `$1`: the full path to the installation package; for example:

  `/Volumes/Projects/Testing/Simple_Carbon_App.pkg`

- `$2`: the full path to the installation destination; for example:

  `/Applications`

- `$3`: the mountpoint of the destination volume; for example:

  `/` or `/Volumes/External_Drive`

- `$4`: the root directory for the current System folder:

  `/`

Environment Variables

When Installer runs an executable file, the file may have access to several environment variables. These variables are discussed next.

Possible Environment Variables

These are the environment variables that a file may have access to when executed:

- `$INSTALLER_TEMP`: The scratch area used by Installer for its temporary work files. This variable is always set. Executable files may use this area to write data, but must not overwrite any Installer files. This directory is erased at the end of installation. For example:

  `/tmp/.Simple_Carbon_App.pkg.897.install`

- `$PACKAGE_PATH`: The full path to the installation package. Same as the `$1` argument. For example:

  `/Volumes/Projects/Testing/Simple_Carbon_App.pkg`

- `$RECEIPT_PATH`: The full path to the directory containing the file being executed. This is a location to which the package has been copied. The location may vary from installation to installation. The executable can currently use this path to locate other files in the package, but this may change in the future. For example:

  `/tmp/.Simple_Carbon_App.pkg.897.install/Receipts/Simple_Carbon_App.pkg/Contents/Resources`

- `$SCRIPT_NAME`: The name of the file being executed. For example:

  `preflight` or `postinstall`

- `$TMPDIR`: This variable is set when the user is doing a Net or CD installation, but is not set when the user is running on a local writable file system. If set, it contains a path to a location on a writable destination volume.

The section that follows lists executable files and the environment variables they can access.

Which Environment Variables a Script Can Access

The following list shows which scripts have access to which environment variables:

- `preflight`:

  No environment variables are available.

- `preinstall`, `preupgrade`:

  `$INSTALLER_TEMP`, `$PACKAGE_PATH`, `$TMPDIR`

- `postflight`, `postupgrade`:

  `$INSTALLER_TEMP`, `$PACKAGE_PATH`, `$RECEIPT_PATH`, `$SCRIPT_NAME`, `$TMPDIR`

- `postinstall`:

  `$INSTALLER_TEMP`, `$RECEIPT_PATH`, `$SCRIPT_NAME`

A Sample Postflight Shell Script

Listing 2 shows a simple postflight script that merely echoes the values of its arguments and of the available environmental variables to the Installer Log window in the Installer application. To show that window, choose File > Show Log. You can print the log window contents or save them to a file. You can open the log window at any time before you quit Installer.

Alternatively, you can redirect output to a log file, using script statements like the following:

```
echo "Start postflight script" >> /tmp/scripts.log
```

Note: Although this script has been tested, you should use care in cutting and pasting it from this document. For more information, see "Troubleshooting Packages and Installation" (page 145).

Listing 2 A postflight script that displays arguments and environment variables

```
#!/bin/bash
#
# This postflight script echoes the values of the available
# arguments and environmental variables.
#
echo "Start postflight script"
echo ""
echo "Arguments:"
echo ""
echo "\$1: full path to the installation package"
echo "     $1"
echo "\$2: full path to the installation destination"
echo "     $2"
echo "\$3: mountpoint of the destination volume"
echo "     $3"
echo "\$4: root directory \"/\" for the current System folder"
echo "     $4"
echo ""
echo "Environment variables available to a postflight executable:"
echo "     INSTALLER_TEMP, PACKAGE_PATH, RECEIPT_PATH, SCRIPT_NAME, and TMPDIR"
echo ""
echo "\$INSTALLER_TEMP: scratch area used by Installer for temporary work files"
echo "     $INSTALLER_TEMP"
echo ""
echo "\$PACKAGE_PATH: full path to the installation package; should be same as \$1"
echo "     $PACKAGE_PATH"
echo ""
echo "\$RECEIPT_PATH: full path to directory containing the file being executed"
echo "     $RECEIPT_PATH"
echo ""
echo "\$SCRIPT_NAME: name of the file being executed"
echo "     $SCRIPT_NAME"
echo ""
echo "\$TMPDIR: if set, a path to a location on a writable destination volume"
echo "     $TMPDIR"
echo ""
echo "End postflight script"
exit 0
```

Listing 3 shows the Installer Log window output from this script for one installation. Note that the output does not contain a value for `$TMPDIR` because that value is only set for a network or CD installation, and this output was generated for a local installation.

Listing 3 Output from postflight script in the Install Log window

```
Start postflight script
Arguments:
$1: full path to the installation package
     /Volumes/Projects/Testing/Simple_Carbon_App.pkg
$2: full path to the installation destination
     /Volumes/External_12_GB/Temporary Items
```

```
$3: mountpoint of the destination volume
     /Volumes/External_12_GB
$4: root directory "/" for the current System folder
     /
Environment variables available to a postflight executable:
     INSTALLER_TEMP, PACKAGE_PATH, RECEIPT_PATH, SCRIPT_NAME, and TMPDIR
$INSTALLER_TEMP: scratch area used by Installer for temporary work files
     /Volumes/External_12_GB/.Simple_Carbon_App.pkg.1085.install
$PACKAGE_PATH: full path to the installation package; should be same as $1
     /Volumes/Projects/Testing/Simple_Carbon_App.pkg
$RECEIPT_PATH: full path to directory containing the file being executed
     /Library/Receipts/Simple_Carbon_App.pkg/Contents/Resources
$SCRIPT_NAME: name of the file being executed
     postflight
$TMPDIR: if set, a path to a location on a writable destination volume

End postflight script
```

Troubleshooting Packages and Installation

This section provides tips for working with executable script files, as well as a question-and-answer section to help troubleshoot common problems.

Important: This section is still under construction. However, it contains some information that can be of use.

For additional information that may aid in troubleshooting, see "Test the Package" (page 49).

Frequently Asked Questions

The following are some frequently asked questions in troubleshooting installation packages.

I changed my package, so why don't the changes show up in Installer?

This happens very commonly when an installation is canceled during testing, but you don't quit from Installer. In this situation, Installer remains open with its current (invalid) state. As a result, you can modify the package, double-click it to launch Installer again, and see the same error because Installer is still running with its saved state.

A similar situation can occur when you modify a package definition file (`.pmsp`) to use a different `Package_contents` directory. If you don't modify the save location, when you re-create the package it will still be in a previous directory. As a result, you may be launching an old package in the current directory, rather than the modified package in some other directory.

Why won't my package install in Mac OS X Version 10.1?

Packages created with the version of PackageMaker that shipped with the Mac OS X version 10.2 (Jaguar) Developer Tools will work only with Jaguar and later versions of the system. However, packages created with PackageMaker 1.1.10 (which first shipped with the December 2002 Developer Tools CD) will work with any version of Mac OS X. For more information, see "Limitations of PackageMaker and Installer" (page 40).

Why doesn't my background picture appear?

Case is important, so be sure your image file is named `background.ext`, where the extension is one of the following: `.jpg`, `.tif`, `.tiff`, `.gif`, `.pict`, `.eps`, and `.pdf`.

Your background image may not appear because it's in the wrong place (for example, you put it in the `Content` directory instead of the `Resources` directory). Or it may not appear because you supplied some localized images, but not one for the user's current Languages preference. To avoid the latter problem, you can place a default image at the highest level in the `Resources` directory.

Localization behavior is similar, but not identical, for the welcome, Read Me, and license files. For details, see "Localizing the Welcome, Read Me, and License Files" (page 63).

A user deleted my software, but still can't reinstall it

Installer looks for a receipt with the same name as your current installation package (for example `MyApp.pkg`) in `/Library/Receipts`. If the receipt is still present, the user may not be able to reinstall the software.

Why doesn't my metapackage work?

A common reason your metapackage doesn't work is that the packages it contains are not in the location it expects them to be. For more information, see "Anatomy of a Metapackage" (page 75).

Why does my package fail in Mac OS X version 10.1 but not in version 10.2?

A package created on Mac OS X version 10.2 with PackageMaker 1.1.10 may fail to install on Mac OS X version 10.1 if the package requires Admin Authorization (which it might require, for example, to install an application in `/Applications`). The install will not authenticate and will halt with the message "You do not have enough privileges to install..."

PackageMaker 1.1.10 creates an `Info.plist` in the package's `Contents` directory, as well as a `<PackageName>.info` file in the package's `/Contents/Resources` directory. The `<PackageName>.info` file provides information needed to install on Mac OS X version 10.1, but in this case, the information is wrong, because the key `NeedsAuthorization` has the value `NO` instead of `YES`.

When this happens, you can fix your package to request authorization, and to install correctly, on Mac OS X version 10.1 by editing the `<PackageName>.info` file and changing the value of the key to `YES`. You can examine the contents of a package in the Finder by Control-clicking it and choosing Show Package Contents from the contextual menu. Once you've located the file in the `/Contents/Resources` directory, you can edit it with a text editor.

Why is Installer ignoring the version.plist information in my package?

This can happen with PackageMaker 1.1.10, because it incorrectly adds the package's Default Location to paths in the `BundleVersions.plist` file (which is described in "The BundleVersions.plist File" (page 108)). Your package will only have a `BundleVersions.plist` file if the package includes software that contains version information in a `version.plist` file, as described in "Supplying Version Information for Your Software" (page 107).)

Once you have created a package with PackageMaker, you can check for this problem, and correct it if necessary, with the following steps:

- In a Terminal window, use the `find` command to find the `Archive.bom` file and the `BundleVersions.plist` file (which you'll use in a later step) in the package:

  ```
  find /Volumes/HD/MyPackage.pkg
  ```

  The output of this command is a listing of all the items in the package. The lines for the two files might look something like this:

  ```
  /Volumes/HD/MyPackage.pkg/Contents/Archive.bom
  ```

  ```
  /Volumes/HD/MyPackage.pkg/Contents/Resources/BundleVersions.plist
  ```

- In a Terminal window, use the `lsbom` command to show, from the package's `Archive.bom` file, all the paths for the items to be installed. (The `Archive.bom` file is described in "High-Level Package Structure" (page 69).)

  ```
  lsbom -p MUGsf '/Volumes/HD/MyPackage.pkg/Contents/Archive.bom'
  ```

  Depending on the package, this may result in a large listing of files, each shown with its path. Among the lines, you should see at least one like the following:

  ```
  drwxrwxr-x root admin ./MyApp.app/Contents/version.plist
  ```

- You can display the contents of the `BundleVersions.plist` file with a command like the following (using the path you obtained in a previous step):

  ```
  cat /Volumes/HD/MyPackage.pkg/Contents/Resources/BundleVersions.plist
  ```

  The resulting output should show a structure of dictionaries and keys, including one or more lines showing a path for the `version.plist` file:

  ```
  <key>./MyApp.app/Contents/version.plist</key>
  ```

  The path should be the same as the paths in the `Archive.bom` file. However, if it looks like the following (that is, it has extra information in front of the expected path), then the `BundleVersions.plist` file is incorrect.

  ```
  <key>./Applications/MyApp.app/Contents/version.plist</key>
  ```

- To correct this problem, edit the `BundleVersions.plist` file and change any paths for the `version.plist` file to match the paths in the `Archive.bom` file. To do so, you delete the part of the path that comes from the Default Location, which in this case is `/Applications`.

Working With Script Files

This section provides some tips for working with script files or other executable files. The primary documentation for these files is located in "Checking the Installation With InstallationCheck" (page 125), "Checking the Installation Volume With VolumeCheck" (page 133), and "Modifying an Installation With Scripts" (page 139).

The following tips address some common issues.

Make sure the script is executable

If you assemble a package with PackageMaker, it will automatically set correctly named scripts to be executable. If you modify a package file directly, it's your responsibility to ensure the executable bit is set.

And while you're at it, make sure your script has adequate permissions (usually at least 555). For more information, see "Setting File Ownership and Permissions" (page 80).

Don't get control characters in the script file

If you edit a script file with a GUI editor, it may insert control characters that interfere with proper execution of the script.

Quote script arguments and environment variables when you use them in a script

The paths and filenames obtained from arguments and environment variables often include spaces, so it's a good idea to quote them in your scripts. For example, without the quotation marks around `$path` in the last line shown in Listing 1, the script could get an error for a volume name that included a space.

Listing 1 A partial VolumeCheck script that shows quoting

```
path=$1
# (Some steps omitted.)
if ([ "$path" = "Jaguar" ]) then
```

This full script is shown in Listing 3 (page 138).

Spell script names correctly

Case matters, so be sure you match the required names (such as VolumeCheck) exactly.

Look for output in the right places

Output from InstallationCheck and VolumeCheck scripts is redirected to the Installer Log window. Output from other installation scripts is redirected to the Console window. Some scripts redirect their own output, such as to a log file. For more details, see "Viewing the Installer Log and Script Output" (page 50).

Specify a shell

Make sure the first line of your shell script specifies a shell, such as:

```
#!/bin/sh
```

Make sure your script returns a value

For success, use `exit 0`. For an error return, use `exit <errorNumber>`.

Make sure you quit Installer between tests

This one needs repeating. If an installation is canceled during testing but you don't quit from Installer, Installer remains open with its current (invalid) state. As a result, you can modify a script, recreate the package, double-click it to launch Installer again, and see the same error because Installer is still running with its saved state.

Document Revision History

This table describes the changes to *Software Distribution Legacy Guide*.

Date	Notes
2006-07-24	Made into legacy document.
	In "Keys That Can Appear in Any Package" (page 89), added note that Installer ignores any values supplied for the `IFMajorVersion` and `IFMinorVersion` keys. (See Installer Release Notes for the latest information on changes in Mac OS X v10.4.)
	In same section, updated the description for the `CFBundleIdentifier` key.
	"Search Methods" (page 100) now describes what happens if you use the Find File mechanism and the user has more than one copy of the software to be updated.
	"Getting Software to the User" (page 17) now distinguishes between the Disk Utility and Disk Copy applications and explains how to view the Disk Utility Help.
	"Distributing Software With Internet-Enabled Disk Images" (page 21) notes that Disk Copy was merged with Disk Utility in Mac OS X version 10.3.
	In description of `IFPkgFlagInstallFat` flag, noted that with the default value of NO, `.o` or `.a` files installed with a package will be thinned along with any other binaries (in "Keys That Can Appear Only in Single Packages" (page 91)).
	In description of variables for installation check files, noted that the values for the variables `$2`, `$3`, and `$4` cannot be trusted (in "Arguments and Environment Variables" (page 127)).
2004-05-27	Modified descriptions (in several places) of when a user can choose a location for packages in a metapackage. See, for example, "Choosing the Location for a Metapackage" (page 49).
	Added the section "Modify the Package for Special Processing" (page 49). It includes some information previously in another section, plus a new link to "Specifying Installation Requirements" (page 111).
	Added the section "Obtaining Owner and Group Information" (page 81). This covers the information from Technical Q&A QA1285.
	Added the trouble-shooting section "Why does my package fail in Mac OS X version 10.1 but not in version 10.2?" (page 146). This covers the information from Technical Q&A QA1283.

Date	Notes
	Added the trouble-shooting section "Why is Installer ignoring the version.plist information in my package?" (page 146). This covers the information from Technical Q&A QA1281.
	Made minor spelling corrections.
2003-08-21	Updated documentation to reflect new and revised features in Mac OS X version 10.3 (v10.3).
	Added "Specifying Installation Requirements" (page 111), which describes a new mechanism available in Mac OS X v10.3 for specifying installation requirements.
	Added a list of supported localization folder names in "Localized Folder Names" (page 70) (not specific to Mac OS X v10.3).
	Revised "Authorization, File Ownership, and Permissions" (page 79), which was formerly called "Authorization and File Ownership".
	Now includes additional information on default ownership and permissions.
	Updated "Finding Previously Installed Software" (page 97):
	Added descriptions for new search methods:
	"BundleVersionFilter" (page 103)
	"CommonAppSearch" (page 104)
	Added examples and updated various descriptions.
	Updated "PackageMaker and Installer Features" (page 35) for new and changed features in Mac OS X v10.3.
	Modified "Other Keys You Might Want to Modify" (page 96) to list new options for:
	specifying installation requirements by modifying a package's information property list;
	specifying that an installation requires a logout by modifying a package's information property list
2003-05-13	First version of Software Distribution released.