Installer Engine 4.5.2
Technical Guide

Document Version 1.4 - December 4, 1998

CHAPTER 1

Installer Script Overview

An **Installer script** encapsulates the presentation of the installable features of a software component to the user, as well as the actions required to install or remove each user-selectable feature. A **software component** can be a complete stand-alone application, or a module intended to be installed in conjunction with other software components.

This document covers the design process and technical details of writing an Installer script for version 4.5.2 of the Installer Engine application. **Installer Engine** is a background-only application that interprets and executes an Installer script, allowing a client application to present the installable features to the user and perform the installation or removal of the selected features. An example of a **client application** is the Upgrader application, which Apple uses as the assistant-like user interface for installing Mac OS and is also licensable by developers. Since Installer Engine provides an extensive Apple event suite for interacting with the Installer script, developers and administrators can write specialized client applications using AppleScript or a programming environment such as MetroWerks or MPW.

Related reading:

- *Cappella User Manual2 D1* – Guide for creating Installer scripts using Cappella, a visual environment for creating and editing Installer scripts using drag-n-drop simplicity.
- *Upgrader 1.2.3 Guide* – Manual for using the client application "Upgrader" as the user experience for your Installer script.
- *ScriptCheck 4.2 User's Guide* – Information about how to use the ScriptCheck MPW tool to check the integrity of your Installer script when using MPW to create your Installer script.

Installer Scripting Writing Process

At its most basic level, the Installer script is a list of instructions for Installer Engine, in the form of resources. Some of these resources define user-visible elements, and some define the low-level actions carried out during the installation.

Choosing an Editing Environment

At the moment, you have three primary methods of creating new Installer script, or modifying an existing Installer script: a user-friendly editing environment called Cappella, the popular resource editing tool Resorcerer , or the more intensive method using MPW.

Cappella — This tool provides a visual environment for creating and editing an Installer script using drag-n-drop simplicity to quickly get your user interface and file actions defined. Cappella provides access to the most popular features of Installer Engine, but if you find you need access a feature Cappella doesn't support, you can easily switch to MPW to finish your Installer script. You'll find the latest version of Cappella on the Installer SDK.

Resorcerer — This third-party resource editing tool combined with the Installer script template file provided on the Installer SDK provides a semi-visual method of creating and editing the resources in an Installer script. You'll still need to understand the relationship between the various resources, but you won't have to learn how to use MPW.

MPW — Apple's Macintosh Programmer's Workshop has been the primary method of creating Installer scripts since the inception of the Apple Installer. Unfortunately, using MPW requires knowledge of the Rez language and an understanding of its tools and environment. Given that MPW provides complete access to all of Installer Engine's features, this document presents its technical information using the MPW form.

Most likely you are learning about Installer scripts for one of two reasons: you need to create a new Installer script, or you need to modify an existing one. Unless you are already familiar with MPW and the Rez language, it's probably best to begin with Cappella. And since Cappella can import existing Installer scripts there's no barrier to prevent switching to Cappella if MPW was used for prior versions of your Installer script. Even if you choose to use MPW or Resorcerer to begin a script from scratch, it might helpful to begin with one of our examples on the SDK or a similar Installer script.

Before You Begin

Given the years of experience we've had at Apple writing Installer scripts for our own products, it's no coincidence that the layout of this technical guide parallels the strategy we use for our largest projects. Since many decisions must be made before the Installer scriptwriter even begins, knowing which questions to ask up front should help reduce changes later in the project. Consider finding answers to this these questions before you begin:

- Does this software component have optional features that the user should be able to choose to install or not install? This will determine if the user must customize the installation, and thereby require you to provide a custom install mode.
- If the user can customize the installation, what are the installable features? Try to define the installable features as important, user-identifiable attributes of the software component. Unless your software component is extremely complex, try to limit the number of features at any one level to five or less. Have the project team agree to the custom install hierarchy before you begin writing your Installer script.

- Which features will be installed by default? This will determine which features are installed as part of the recommended feature set. You should always provide a recommended installation (easy install).
- Is the recommended installation dependent on certain system software or hardware attributes or conditions? This will require you to write additional Installer rules to make decisions at run time. Compare the attributes you must inspect with the built-in functionality of Installer Engine to predict whether you will have to write rule function code resources.
- Are there other collections of optional features that users will want to install? If your software component has many optional features, providing additional feature sets, such as “Minimal Installation” or “Full Installation”, will help users quickly select the configuration that is best for them.
- What files must be installed for each user-selectable feature? Knowing all the files you’ll eventually install up-front is very helpful, even if you must install dummy files until the real ones are available.
- Which pieces can’t simply be installed as complete files? You’ll what to know about any special situations that cause you to create new files or update existing files using a method other than a simple file copy. You might need to learn how to install or remove individual resources using Resource Atoms.
- Which files are shared with other features or other software that cannot be removed? You’ll want to prevent any shared files, such as libraries, system software pieces, etc. from beging removed if you allow the user to remove the feature.
- Can you depend on files being in predetermined locations? If you must search for files to update or remove, you’ll need to write a code resource to perform the actual search.

Getting Started

Once you’ve found some answers to our recommended questions you’re ready to begin creating the necessary resources the will make up your Installer script. If you are:

- Using Cappella to create or modify your Installer script, it will be best if you use the Cappella manual to as your primary guide and reference this document whenever you need additional detail.
- Using MPW or Resorcerer to create or modify your Installer script, use this document.

It’s best if you start by creating a skeletal implementation of your user interface. This means creating the resources that define the custom install feature hierarchy, the recommended installation features set, plus additional feature sets you’ve defined. The chapter “Defining the User Interface” describes the process of implementing the user interface.

Next, you’ll need to create the resources that describe the actions to perform. The chapters “Define Action” and “File Specification” will give you the information you need to describe the actions to perform to install each feature.

Next, there are some miscellaneous resources that must be added to your script to make it complete.

Finally, you’ll want to run the ScriptCheck MPW tool on your Installer script to check the integrity of your Installer script and update size and date information. See *ScriptCheck 4.2 User’s Guide* for more information about using ScriptCheck.

Additional chapters cover the API available to code resources, run-time issues, and using Apple events to drive Installer Engine.

CHAPTER 2

Defining the User Interface

This chapter is broken into two main sections. First, we walk you through the process of defining the user interface of your Installer script, then we provide a reference section for each Installer script resource related to implementing the user interface. We use the "User Interface Example" from the Installer SDK, for this and many of the subsequent chapters.

Example (Part 1): Implementing the User Interface

Our goal is to create a custom install feature list with multiple levels, and three easy feature sets — recommended, minimal, and full — that install various collections of features. Although we don't write any decision making code using Installer rules in this example, you'll find several other examples on the SDK that do. Let's start, why don't we!

Step 1: Create the Custom Install Feature Hierarchy

Image your project team has decided they want the following feature hierarchy for your software component:

		Feature ID	Removable
[ ] Feature #1	(i)	1001	Yes
[ ] Feature #2	(i)	1002	No
----------		9999	-
[ ] More Features		1003	Partial
[ ] Feature #3		2001	Yes
[ ] Even More Features		2002	Partial
[ ] Feature #4		3001	No
[ ] Feature #5		3002	Yes
[ ] Feature #6		1004	No
[ ] Feature #7		1005	Yes

We've arbitrarily assigned IDs to each feature, and indicated whether the feature can be removed. Although only two of the features are specified to have information buttons ((i)), you should normally supply information buttons for all features.

To define each feature, we create a package resource ('inpk') with our chosen ID and the feature name we wish to be displayed to the user. We describe the package resources using the Rez language in the following way:

```
resource 'inpk' (1001) {
    format0 {
        showsOnCustom,       // show the package as a selectable item
```

```
                            // item when used as a subpackage

        removable,          // show under Custom Remove as a selectable
                            // item, when package is a subpackage

        dontForceRestart,   // don't make user reboot after installation

        1001,               // package comments resource ID

        0,                  // package size ( if 0, filled by ScriptCheck )

        "Feature #1",       // Custom Install selection description
        {
            'infa', 1001;       // file to install or remove
        },
    }
};

resource 'inpk' ( 1002 ) { format0 { showsOnCustom, notRemovable, dontForceRestart,
1002, 0, "Feature #2", { 'infa', 1002 }, } };

resource 'inpk' ( 9999 ) { format0 { showsOnCustom, notRemovable, dontForceRestart,
0, 0, "-", {}, } };

resource 'inpk' ( 1003 ) { format0 { showsOnCustom, removable, dontForceRestart,
0, 0, "More Features", { 'inpk', 2001,'inpk', 2002 }, } };

resource 'inpk' ( 2001 ) { format0 { showsOnCustom, removable, dontForceRestart,
0, 0, "Feature #3", { 'infa', 2001 }, } };

resource 'inpk' ( 2002 ) { format0 { showsOnCustom, removable, dontForceRestart,
0, 0, "Even More Features", { 'inpk', 3001,'inpk', 3002 }, } };

resource 'inpk' ( 3001 ) { format0 { showsOnCustom, notRemovable, dontForceRestart,
0, 0, "Feature #4", { 'infa', 3001 }, } };

resource 'inpk' ( 3002 ) { format0 { showsOnCustom, removable, dontForceRestart,
0, 0, "Feature #5", { 'infa', 3002 }, } };

resource 'inpk' ( 1004 ) { format0 { showsOnCustom, notRemovable, dontForceRestart,
0, 0, "Feature #6", { 'infa', 1004 }, } };

resource 'inpk' ( 1005 ) { format0 { showsOnCustom, removable, dontForceRestart,
0, 0, "Feature #7", { 'infa', 1005 }, } };
```

Each package resource contains, either a list of actions perform, or a list of sub features. For our example, all leaf node features reference a single file copy action (‘infa’), and the features which contain sub features, reference other package resources (‘inpk’).

Step 2: Add Feature Information Resources

To help the user choose the most appropriate features when customizing the installation, you should provide extra information for each feature. This is accomplished by referencing a package comment resource (‘inpc’) from each package resource (‘inpk’).

The feature information resources in the Rez language:

```
resource 'inpc' ( 1001 ) {
    format1 {
        0,                  // sample date ( 08/08/94 seconds since 1904)
        0x08018000,             // sample version ( 8.0.1 GM)

        0,                  // Ignored, not shown in user interface
```

```
        9128,                   // icon resource ID ( 'ICN#', 'icl4', 'icl8' )
                                // - ID must be greater than 1024
                                // - resource item is in included rsrc file

        1001                    // 'TEXT' resource ID of item containing package description
    }
};

resource 'inpc' ( 1002 ) { format1 { 0, 0, 0, 9128,   1002 } };

data 'TEXT' ( 1001 ) { "This feature installs 'Example File • 1'." };
data 'TEXT' ( 1002 ) { "This feature installs 'Example File • 2'." };

// resource 'icl8' ( 9128 ) - Not listed due to space, see “UserInterfaceExample.r” for resource.
```

Step 3: Create the Custom Install Framework

To tell Installer Engine which package resources make up our feature hierarchy, we must create a framework which adds the top-level features to the list. Since our framework doesn't make any decisions, it references a single rule resource that calls the AddCustomItems rule clause.

Here's what these resources look like in the Rez language:

```
// custom install framework always uses ID of 766
resource 'infr' (766) {
    format0 {{
        pickFirst, { 800 },
    }}
};

// rule that adds top-level features to Custom Install
resource 'inrl' (800) {
    format0 {{
        AddCustomItems{{ 1001, 1002, 1003, 1004, 1005 }},
    }}
};
```

Step 4: Create the Recommended Feature Set Framework

For users that don't want to customize the installation of our software component, but instead wish to let the installation program decide what's best we must define the recommended installation. To do this we create another framework that adds those features we want installed. We also provide a prompt string to be displayed when this feature set is chosen to help guide the user.

Let's assume your project team has decided that the following features will be installed as part of the recommended installation: “Feature #1”, “Feature #3”, “Feature #5”, and “Feature #7”. Here's what these resources look like in the Rez language:

```
// Recommended Feature Set
resource 'infr' (1000) {
    format0 {{
        pickFirst, { 1000 },
    }}
};

// Rule that specifies Recommended Feature Set
resource 'inrl' (1000) {
    format0 {{
        // define recommended feature set and user prompt
        AddPackages{{ 1001, 2001, 3002, 1005 }},
        AddUserDescription{ "Click Start to install the recommended software onto “^0”." },
```

```
    }}
};
```

Step 5: Create Additional Easy Feature Set Frameworks

In addition to the recommended feature set, the project team has also decided to provide the user with pre-defined feature sets for both minimal and full installations. The full feature set will contain all features, and the minimal feature set will contain the following features: “Feature #1” and “Feature #5”. To do this we create an additional framework for each feature set.

Here’s what these resources look like in the Rez language:

```
// Full Feature Set
resource 'infr' (1001) {
    format0 {{
        pickFirst, { 1001 },
    }}
};

// Rule that specifies Full Feature Set
resource 'inrl' (1001) {
    format0 {{
        // define ‘full’ feature set and user prompt
        AddPackages{{ 1001, 1002, 1003, 1004, 1005 }},
        AddUserDescription{ "Click Start to install all software onto “^0”." },
    }}
};

// Minimal Feature Set
resource 'infr' (1002) {
    format0 {{
        pickFirst, { 1002 },
    }}
};

// Rule that specifies Minimal Feature Set
resource 'inrl' (1002) {
    format0 {{
        // define ‘minimal feature set and user prompt
        AddPackages{{ 1001, 3002 }},
        AddUserDescription{ "Click Start to install a minimal set of software onto “^0”." },
    }}
};
```

Step 6: Create a Preference Resource

Installer Engine provides a default preference resource, but because we want additional easy feature sets, we reference these additional framework resources from our own preference resource.

The preference resource define in the Rez language:

```
resource 'inpr' (300) {
    format1 {
        useDiskTargetMode,       // User chooses an entire disk as the destination.
        noSetupFunctionSupplied, // Not using a setup function code resource
        dontAllowCleanInstall,   // Clean install is only appropriate when installing Apple SSW
        isNotSSWInstallation,    // We're not installing System Software
        '',                      // Setup function type, but we don't have one so it's zero
        0,                       // Setup function ID, but we don't have one so it's zero
        0,                       // Text Encoding ID of product's localized language, 0 for U.S.
        1000,                    // ID of 'STR#' resource to store feature set names
        {
```

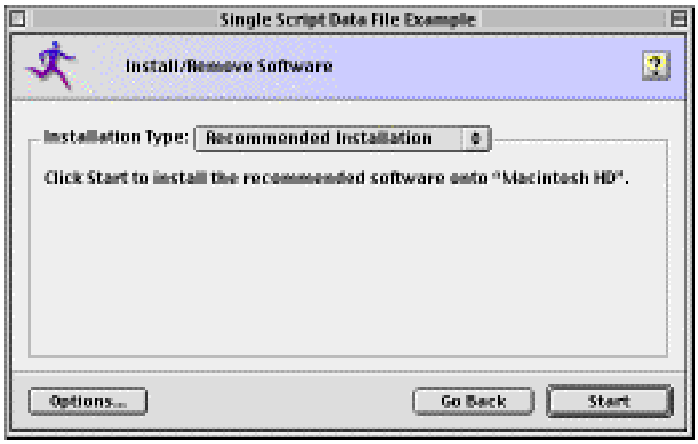
```
            1000, 1,                // Recommended Feature Set (Feature set rule framework ID and
                                    // string index of feature set name)
            1001, 2,                // Full Feature Set
            1002, 3,                // Minimal Feature Set
        },
        ""                          // Default target folder name.  Not needed for disk mode.
    }
};

// Feature Set names as presented to the user
resource 'STR#' (1000) {
    {
        /* [1] */  "Recommended Installation",
        /* [2] */  "Full Installation",
        /* [3] */  "Minimal Installation",
    }
};
```

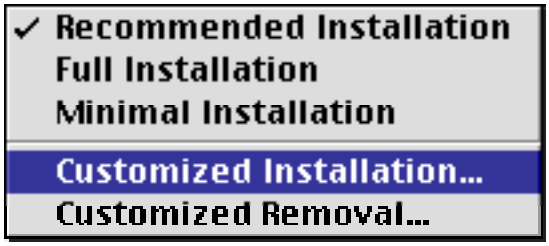
Step 7: Test the User Interface

Although we won't be able to install anything, if we Rez the resources we've defined so far and create an Installer script file, we can see our user interface in action. Using the single Installer script mode of Upgrader, we should see the following interface:

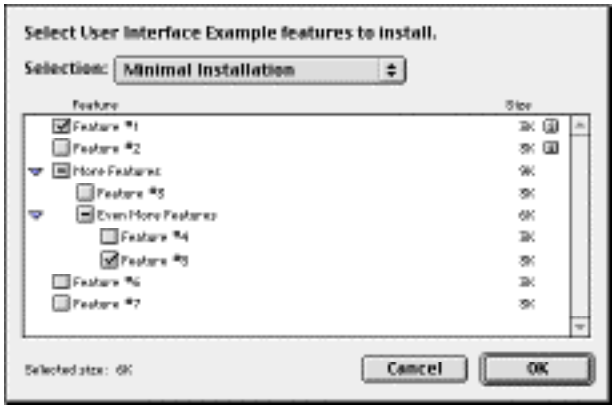
The recommended installation is the initially selected feature set when first entering the installation panel.



All feature sets are shown when the user clicks the Installation Type pop-up menu, in addition to items for custom install and remove.



Selecting Customized Installation from the pop-up presents the Custom Install Selection Dialog, in which the user has chosen the minimal installation feature set.



In the next chapter the example will continue with definition of the actions.

Using Packages (‘inpk’)

In addition to defining the custom install feature hierarchy, package resources can also be used to group atoms together to make your scriptwriting easier. Installer Engine only knows what to install by the package resource IDs you pass to it via the AddPackages and AddCustomItems rule clauses. The package resource contains a list of those atoms or other packages that you have specified to be installed or removed when the parent package is installed or removed.

Atom Execution Order

Once the user starts the installation or removal, all specified packages are decomposed into a flat list of atoms. Therefore, the order Installer Engine executes the atoms is unrelated to their position in the package resource’s part list. Prior to installation Installer Engine groups File, Resource, Font, ResMerge, and Folder Atoms based on the source disk they reference. Those atoms that do not need source disks (atoms that delete on install) are grouped together.

For each source disk Installer Engine executes the atoms in the following order:

- Folder Atoms.
- File Atoms. This allows you to overwrite a file copied with a Folder Atom on the same or previous source disk.
- ResMerge Atoms. This allows you to add or replace additional resources to a file copied on the same or previous source disk.
- Resource Atoms. This allows you to add or replace a resource in a file copied with a Folder, File or ResMerge Atom on the same or previous source disk.
- Font Atoms.

Although Installer Engine will consistently order the atoms of a specific type, the scriptwriter should never depend on this ordering, except for Action Atoms, which are always guaranteed to be ordered by their ‘inaa’ resource ID.

To ensure that the source disks will always be copied in the same order, include a Disk Order resource (‘indo’) in your script.

Package Reference

Package Resource Description

Packages have an 'inpk' resource type. Its template is shown below.

```
#define PackageFlags                          \
    boolean doesntShowOnCustom, showsOnCustom;  \
    boolean notRemovable, removable;     \
    boolean forceRestart, dontForceRestart;  \
    fill bit[13]   /* Reserved */

type 'inpk' {
    switch {
        case format0:
            key integer = 0;             /* Package Format version */
            PackageFlags;                /* Package Flags */
            unsigned integer;            /* Package Comment Rsrc ID */
            unsigned longint;            /* Package Size */
            EvenPaddedString;            /* Package Name */
            unsigned integer = $$CountOf(PartsList);
            wide array PartsList {       /* Parts List */
               RsrcType;                 /* Part Type */
               RsrcID;                   /* Part ID */
            };
    };
};
```

Flag descriptions

`showsOnCustom/doesntShowOnCustom`
: Determines if the package should be displayed in the Custom Install list.

`removable/notRemovable`
: Determines if the package should be added to the list of packages to be removed when the user removes this feature. When used with the showsOnCustom flag, determines if the package is selectable in the Custom Remove list. Note that you must also set the `deleteWhenRemoving` flag on any File, Resource or Font Atoms that are to be removed and you must set the `actOnRemove` flag in any Action Atoms that are to be called.

  To include a package as a selectable item within Custom Remove, it is necessary to also include the package as a selectable item in Custom Install. There is no option to display a package as a selectable item in Custom Remove, but not in Custom Install. You may however choose to do nothing if the package is a selected item within Custom Install, and only act on that selected package during Custom Remove, but this may be confusing to the user.

`forceRestart/dontForceRestart`
: Determines if the client application should force the user to restart their Macintosh after installing or removing this package when the currently active System Folder is a target. Use the `forceRestart` flag if the installation

requires rebooting to gain functionality, or if the package makes changes to the System file.

Field descriptions

`Package Comment Rsrc ID`

The resource ID of a package comment resource that displays the package's size, date, version and a brief description of its contents or what it does. It's recommended that every package shown in the Custom Install list have a comment resource. If you do not wish to provide a package comment, place a zero in this field. There are two types of comment resources. The first, an 'icmt' resource, provides basic information in one resource. The newer 'inpc' resource supports the RAM Size field and references a 'TEXT' resource that allows up to 32K of information text. (2-bytes)

`Package Size`

The size of all atoms and packages contained in this package. This value is used to display in the comment window as well as provide the estimated selection size provided in Custom Install. Installer Engine does not use this field to determine the actual disk space requirements of the target hard disk during preflighting. Individual atom sizes are summed to determine if sufficient target disk space is available. (4-bytes)

`Package Name`

The name to be displayed in the Custom Install list. (even-padded Pascal string)

If the Package Name field is a single dash character (-), a gray horizontal line appears in the list of packages. This dash line is similar to the one found for menu resources. This line can be used to separate the items that can be installed into logical groups. If a package is used just for displaying a separation line, all of the other fields are ignored.

Below is a sample gray-line package:

```
resource 'inpk' (1, "dashed line") {
   format0 {
         showsOnCustom,
         notRemovable,
         dontForceRestart,
         0,    /* Pkg Cmnt Rsrc ID */
         0,    /* Package size */
         "-",  /* Package Name */
         {} /* empty brackets */
   }
};
```

`Part List`

The type and resource ID of each part contained in this package. (4-bytes + 2-bytes for each list item)

Package Comment Resource Description

Visible packages should reference one of two comment resources to provide additional information to the user about the packages they might want to install or remove. The older 'icmt' is supported, but cannot specify the RAM Size field. The new 'inpc' comment resource must reference a 'TEXT' resource that contains the text to display at the bottom of the package info window. Both templates are described below.

New Package Comment Resource Description

```
type 'inpc' {                          * New Custom Item comment */
       switch {
           case format1:
               key integer = 1;         /* Custom Item Format version */
               unsigned hex longint; /* Custom Item Date */
               unsigned hex longint; /* Custom Item Version */
               unsigned hex longint; /* Custom Item RAM Requirements */
               rsrcID;                  /* Custom Item Icon ID */
               rsrcID;                  /* Custom Item Desc. ('TEXT' ID)*/
       };
};
```

Field descriptions

`Custom Item Date` — The release date of the package. The value is specified in seconds since January 1, 1904. Installer Engine uses the IUDateString toolbox call to get the string to display. See *Inside Macintosh Volume II*, page 377 and *Inside Macintosh Volume I*, page 504. (4-bytes)

`Custom Item Version` — The version number of this package. Installer Engine can decipher two version formats:

Binary coded decimal number. The ones digit is the secondary revision (0 to 9), the tens digit is the primary revision (0 to 9), and all greater digits make up the version number. For example, 100 is version 1.0, 290 is version 2.9, 605 is version 6.0.5, and 5704 is version 57.0.4. Installer Engine knows this format because it will always be less than 10000. (4-bytes)

'vers' resource hexadecimal number. See the definition of 'Version' in the "InstallerTypes.r" file. (4-bytes)

`Custom Item RAM Requirements` — The estimated RAM needs of this package. Installer Engine displays this value in the package information window. (4-bytes)

`Custom Item Icon ID` — The ID of the appropriate `ICON, ICN#, icl4 and/or icl8` resources. The ID must be 1024 or greater (IDs below this range are reserved). If the script contains these icons, the appropriate one is displayed in the upper left corner of the package information window. (2-bytes)

`Custom Item Description` — The ID of a 'TEXT' resource containing the text to appear in the comment section of the package information window. It

should describe the package and provide any information that will help the user decide if installing or removing the package is appropriate. Styled text is not currently supported. (2-bytes)

Old Package Comment Resource Description

```
type 'icmt' {
       unsigned hex longint; /* Release Date */
       unsigned hex longint; /* Version Number*/
       RsrcID;                /* Icon ID */
       EvenPaddedString;      /* Package Comment Text */
};
```

Field descriptions

`Release Date`	The release date of the package. The value is specified in seconds since January 1, 1904. Installer Engine uses the IUDateString toolbox call to get the string to display. See *Inside Macintosh Volume II*, page 377 and *Inside Macintosh Volume I*, page 504. (4-bytes)
`Version Number`	The version number of this package. Installer Engine can decipher two version formats: **Binary coded decimal number.** The ones digit is the secondary revision (0 to 9), the tens digit is the primary revision (0 to 9), and all greater digits make up the version number. For example, 100 is version 1.0, 290 is version 2.9, 605 is version 6.0.5, and 5704 is version 57.0.4. Installer Engine knows this format because it will always be less than 10000. (4-bytes) **'vers' resource hexadecimal number.** See the definition of 'Version' in the "InstallerTypes.r" file. (4-bytes)
`Icon ID`	The ID of the appropriate `ICON, ICN#, icl4 and/or icl8` resources. The ID must be 1024 or greater (IDs below this range are reserved). If the script contains these icons, the appropriate one is displayed in the upper left corner of the Package Comment window. (2-bytes)
`Package Comment Text`	The text which appears in the comment section of the Package Comment. It should describe the package and provide any information that will help the user decide if installing or removing the package is appropriate. (even-padded Pascal string)

Using Frameworks ('infr')

The framework and rule resources work together to enable you to define the user interface and make run-time decisions about what should be installed based your environment. For example, in Apple's Mac OS Installer scripts, we install different software depending on the type of machine on which Installer Engine is running, the type of target disk the user has chosen, and what has previously been installed. Depending on the requirements of your user interface, you may need to create some or all of the three uses of framework resources.

Custom Install Rule Framework

The custom install rule framework provides control over items shown in the custom install feature hierarchy. When Installer Engine finds and opens the script, the custom install framework is located by looking for an 'infr' resource of ID 766. If one is not found, Installer Engine uses the Installer 3.X method of determining which packages should be shown in the custom install feature hierarchy to provide compatibility with older Installer scripts.

In order for features to appear in the list, the scriptwriter must call the `AddCustomItem` rule clause for each top-level feature to be displayed. The `AddCustomItem` rule clause accepts a list of the package resource ('inpk') IDs. The items will be listed in the order they are added. Sub-packages with the `showsOnCustom` flag set will automatically be added at the appropriate levels in the hierarchical list.

Easy Feature Set Rule Framework

The easy feature set rule framework provides control over the prompt string and package resources specified to be installed for the feature set. When using the older, format 0 of the preference resource ('inpr'), Installer Engine locates the recommended installation framework by looking for an 'infr' resource that does not have the ID of 765 or 766. This is to provide compatibility with Installer 3.4.X scripts that can use any ID for their 'infr' resource. If you are using the latest version of the preference resource, then you will have listed the IDs of the easy feature sets in the order you wish to display them to the user.

Global Rule Framework

The global framework is an optional framework that provides common rules between the custom install and easy feature set frameworks. The global framework is always executed before the easy feature set or custom install frameworks, thereby encouraging smaller and simpler scripts by allowing the other frameworks to refer to assertions set in the global framework. All rule clauses can be called in the global framework, but those not relevant to the interface mode are ignored. The global framework is located by looking for an 'infr' resource of ID 765.

Using Rules ('inrl')

Rules, which are organized by the rule framework, provide the ability to make run-time decisions. Rules can examine the installation environment, save information about the installation environment to simplify other rules, provide information to the user about what the installation will do for them, report errors, and/or recommend which features should be installed.

Rules are composed of one or more clauses. Each clause returns a Boolean value. The first clause that returns FALSE terminates the rule. If the terminated rule is part of a `pickFirst` set in the rule framework then the following rule, if any, is evaluated. If all of the clauses in a rule return `TRUE`, the rule is said to fire. If the firing rule is part of a `pickFirst` set in the rule framework then no more rules in the set are evaluated.

Clauses can be combined to develop the equivalent of an if-then statement. For example, the if-then statement:

```
IF the chosen target is a floppy disk
THEN use the package named floppyInstall in the Easy Install
```

would be written as a rule as follows:

```
resource 'inrl' (rFloppyInstall) {
    format0 {{
        checkTgtVolSize {floppy, floppy},    <-- A Clause
        addPackages {{pFloppyInstall}}       <-- Another Clause
    }};
};
```

The scriptwriter defines the constants `rFloppyInstall` and `pFloppyInstall` earlier in the script (for example: `#define rFloppyInstall 1000`). This rule has two clauses. The first clause, `checkTgtVolSize` returns `TRUE` if the currently chosen target volume is a floppy. If `checkTgtVolSize` returns `TRUE`, the second clause is evaluated. This second clause, `addPackages`, always returns `TRUE`, but also has the effect of adding the package whose ID is `pFloppyInstall` to the list of packages to be used in the Easy Install. So, if the first clause is `TRUE`, the addPackages clause will be TRUE and the rule will fire.

Clauses can be `AND`ed together. For example, we could write:

```
IF the chosen target is a floppy disk
    AND we are installing on a Mac+
THEN return the package named floppyInstall to Easy Install
    AND add "• Mac+ Floppy Installation" to the Easy Install text
```

as

```
resource 'inrl' (rFloppyInstall) {
    format0 {{
        checkTgtVolSize {floppy, floppy},
        checkGestalt {gestaltMachineType, {gestaltMacPlus}},
        addPackages {{pFloppyInstall}},
        addUserDescription {'• Mac+ Floppy Installation'}
    }};
};
```

This rule has four clauses. If the first two return `TRUE` (i.e., we are installing onto a floppy and the current machine is a Mac Plus) then the second two clauses will be evaluated. Note that the clauses `addPackages` and `addUserDescription` always return `TRUE`.

For a more complete example, suppose you wanted to add a piece of text to the Easy Install message which described the machine you were installing onto. The pseudo code is as follows:

```
IF (Machine=Plus) THEN
    Return the message 'Installing onto your Macintosh Plus'
ELSE IF (Machine=SE) THEN
    Return the message 'Installing onto your Macintosh SE
ELSE IF (Machine=II) THEN
    Return the message 'Installing onto your Macintosh II'
```

The rule framework is as follows:

```
resource 'infr' (kEasyInstallFrameworkRsrcID ) {
    format0 {{
        pickFirst, {rPlusUD, rSEUD, rIIUD},
    }}
};
```

The rules which this framework references are as follows:

```
resource 'inrl' (rPlusUD) {
    format0 {{
        checkGestalt {gestaltMachineType, {gestaltMacPlus}},
        addUserDescription {'Installing onto your Macintosh Plus'}
    }};
```

```
};

resource 'inrl' (rSEUD) {
    format0 {{
        checkGestalt {gestaltMachineType, {gestaltMacSE}},
        addUserDescription {'Installing onto your Macintosh SE}
    }};
};

resource 'inrl' (rIIUD) {
    format0 {{
        checkGestalt {gestaltMachineType, {gestaltMacII}},
        addUserDescription {'Installing onto your Macintosh II}
    }};
};
```

The constants, `rPlusUD`, `rSEUD`, and `rIIUD` would be defined in the script before being used.

Using Assertions

It is often useful to keep track of information about the type of installation that is being done. This allows a more natural script organization. Note: Installer Engine clears all assertions prior to firing rules. Suppose, for example, that many of the rules in a script depended on knowing whether an update or a new install was being done. The pseudo code for this is as follows:

```
IF System File Exists
    AND Installing on Mac+
THEN Use Mac+ Update packages

IF System File Exists
    AND Installing on MacSE
THEN Use MacSE Update packages
```

More conveniently (and with better performance):

```
IF System File Exists
THEN Assert(updating)

IF updating
    AND Installing on Mac+
THEN Use Mac+ Update packages

IF updating
    AND Installing on MacSE
THEN Use MacSE Update packages
```

Assertions provide this functionality. Using assertions is a three-step process: 1) define a unique constant for the assertion, 2) set the assertion using an `AddAssertion` clause, and 3) check the assertions via `CheckAllAssertions`, `CheckAnyAssertion`, `CheckMoreThanOneAssertion`, `CheckAllNonAssertions`, or `CheckAnyNonAssertion` clauses. The above example is written as follows:

```
#define aUpdating 1                                    <-- uniquely defined

resource 'infr' (kEasyInstallFrameworkRsrcID) {
    format0 {{
        pickAll, {rCheckUpdate, rSEUpdate, rIIUpdate},
    }}
};

resource 'inrl' (rCheckUpdate) {
```

```
    format0 {{
        checkFileRsrcForkExists {fsSystemFile},
        addAssertion {{aUpdating}}                  <-- set it
    }};
};

resource 'inrl' (rSEUpdate) {
    format0 {{
        checkAllAssertions {aUpdating},          <-- check it
        checkGestalt {gestaltMachineType, {gestaltMacSE}},
        addPackages {{pSEUpdate}}
    }};
};

resource 'inrl' (rIIUpdate) {
    format0 {{
        checkAllAssertions {aUpdating}           <-- check it
        checkGestalt {gestaltMachineType, {gestaltMacII}},
        addPackages {{pIIUpdate}}
    }};
};
```

Rule Execution

Frameworks are evaluated whenever the environment may have changed. Although this should not be an issue with most Installer script, don't assume your rules are only executed once during the installation.

Supporting Up-Front Custom Feature Selection

To enable the user to customize the installation of your product in future versions of Upgrader, we must run the custom install rules before the installation actually starts. Most Installer scripts warn the user if the target volume will not support the installation, such as a missing System Folder or files. This works fine when the Installer script is run individually, but as a chained install in Upgrader there may be one or more Installer scripts that will be run before your script. If the target disk is completely empty, then most scripts correctly call ReportVolError, but some don't call AddCustomItems in this case. The result is that the human interface doesn't have a custom install hierarchy to show, and consequently can't let the user customize the installation, thereby forcing the user to perform an easy install.

The changes required to support this scenario are quite simple, and many scripts already support this scenario. To solve this problem, we need the scriptwriter to always call AddCustomItems, even when they report a volume error. The human interface code will ignore your volume error and present the custom install hierarchy so the user can choose the features to be installed later. If at the time of installation, your custom rules still returns an error, Upgrader will alert the user and stop the installation of your product.

As an example of a script that doesn't support up-front custom selection in all cases, consider the following custom rules (displayed in InstallTalk pseudo code). These rules make a simple decision based on the version and existence of the target System file.

Listing 3-3 Problematic custom feature set rules

```
if CheckFileVersion( targetFile20000, 8, 0, 0, final, 0 ) then
    AddCustomItems( package1, package3 )
else if CheckFileVersion( targetFile20001, 7, 6, 0, final, 0 ) then
    AddCustomItems( package2, package3 )
else
```

```
    ReportVolError( "This product requires System 7.6 or later." )
end
```

To make the above custom rules provide a custom hierarchy even when calling ReportVolError, we need to add calls to AddCustomItems, as shown in the modified rules below.

Listing 3-4 Problematic custom feature set rules

```
if CheckFileVersion( targetFile20000, 8, 0, 0, final, 0 ) then
    AddCustomItems( package1, package3 )
else if CheckFileVersion( targetFile20001, 7, 6, 0, final, 0 ) then
    AddCustomItems( package2, package3 )
else
    ReportVolError( "This product requires System 7.6 or later." )
    AddCustomItems( package1, package3 )
end
```

With these changes Upgrader can present a feature hierarchy to the user even when the target is empty. Unfortunately, this strategy still presents a problem to Installer when upgrading from 7.6 to 8.X. When we run the custom rules before the installation is started in order for the user to select the desired features, package 2 will be added to feature hierarchy. But when the Installer script is actually run — chained somewhere after the SSW script — the custom rules will add package 1. To combat this problem, Upgrader saves the package name in addition to the package ID so the correct package is selected once the installation is ready to begin. In the example above, it would be important for the scriptwriter to use the same name for both package 1 and package 2.

Note: To Upgrader correctly handle up-front custom selection, set the Support Dynamic Custom Hierarchy option for the software component in the installation plug-in preference resource.

Supporting Easy Feature Sets

Users have always wanted to see which custom features are installed for an easy install. Like the up-front custom selection problem discussed above, we need to have the easy rules call AddPackages even when it calls ReportVolError. Additionally, the package IDs passed in the calls to AddPackages must correspond to the package IDs shown in the custom hierarchy.

To illustrate the problem, consider the following easy rules:

Listing 3-5 Problematic easy feature set rules

```
AddUserDescription( "Click Install to place ..." )
    AddPackages( package4 )
else if CheckFileVersion( targetFile20001, 7, 6, 0, final, 0 ) then
    AddUserDescription( "Click Install to place ..." )
    AddPackages( package5 )
else
    ReportVolError( "This product requires System 7.6 or later." )
end
```

There are two problems here. The first problem is that package 4 or package 5 is added, but neither was added using AddCustomItems, directly or indirectly. To fix this problem we add package 1 or package 2 instead. The second problem is the same problem we had with the custom rules. Since AddPackages doesn't get called when the target System file doesn't exist, the human interface cannot show which features will be installed for an easy install. To fix this, we add calls to AddPackages even when ReportVolError is called.

Listing 3-6 Corrected easy feature set rules

```
if CheckFileVersion( targetFile20000, 8, 0, 0, final, 0 ) then
    AddUserDescription( "Click Install to place ..." )
    AddPackages( package1 )
else if CheckFileVersion( targetFile20001, 7, 6, 0, final, 0 ) then
    AddUserDescription( "Click Install to place ..." )
    AddPackages( package2 )
else
    ReportVolError( "This product requires System 7.6 or later." )
    AddPackages( package1 )
end
```

Since calls to ReportVolError always override any calls to AddCustomItems, AddPackages, or AddUserDescription, you don't have to worry about confusing the Upgrader by calling ReportVolError in addition to calling AddPackages. The modifications presented above will continue to be compatible with older versions of Installer.

Rule Clause Reference

This section describes the available rule clauses.

```
#define kEasyInstallFrameworkRsrcID     764
#define kCustomInstallFrameworkRsrcID   766
#define kGlobalFrameworkRsrcID          765
```

Global Rule Clauses

CheckGestalt

```
CheckGestalt { gestaltSelector, {gestaltReturnValuesList} }
```

`gestaltSelector`	A selector value specifying the type of system information to check. (4-bytes)
`gestaltReturnValuesList`	A list of Gestalt return values. If the Gestalt result matches one of the values in the list the function will return TRUE. (4-bytes)

DESCRIPTION

Use the `CheckGestalt` clause to check a `Gestalt` attribute. `CheckGestalt` takes two arguments: a `Gestalt` selector, and a list of valid `Gestalt` return values. Installer Engine calls `Gestalt` with the given selector. If the `Gestalt` return value is in the list of valid return values, `checkGestalt` returns `TRUE`.

NOTE

If the Gestalt selector specified requires a value containing flags (or a bit mask), you should use the CheckGestaltAttributes routine described below. ◆

CheckGestaltAttributes

```
CheckGestaltAttributes { gestaltSelector, bitMask }
```

gestaltSelector	A selector value specifying the type of system information to check. (4-bytes)
bitMask	A bit mask that will be AND'ed with actual result from the Gestalt call. (4-bytes)

DESCRIPTION

Use the CheckGestaltAttributes clause to check one or more specific bits in the Gestalt value. Passing a bit mask with more than one bit set will effectively OR the specified bits in the Gestalt result value. CheckGestaltAttributes takes two arguments: a Gestalt selector, and a bit mask. Installer Engine calls Gestalt with gestaltSelector, then AND's the result value with bitMask. If this value is non-zero then CheckGestaltAttributes returns TRUE.

CheckMinMemory

```
CheckMinMemory{ minimalMemory }
```

minimalMemory	The minimal number of megabytes of physical memory. (4-bytes)

DESCRIPTION

Use CheckMinMemory to specify the minimal amount of physical memory needed for an installation. Note that you cannot use CheckGestalt for this job unless you want to list *all* of the memory configurations applicable for an installation. CheckMinMemory takes a single argument: the minimal amount of memory (in MB) needed for this clause to be TRUE.

CheckFileDataForkExists

```
CheckFileDataForkExists{ targetFileSpecRsrcID }
```

targetFileSpecRsrcID	A Target File Spec resource ID. (2-bytes)

DESCRIPTION

Use CheckFileDataForkExists to determine whether a specific file has a data fork. It takes a single argument: the Target File Spec ID of the file you are interested in. CheckFileDataForkExists returns TRUE if the file referenced by the File Spec exists and has a data fork.

CheckFileRsrcForkExists

```
CheckFileRsrcForkExists{ targetFileSpecRsrcID }
```

targetFileSpecRsrcID	A Target File Spec resource ID. (2-bytes)

DESCRIPTION

Use CheckFileRsrcForkExists to determine whether a specific file has a resource fork. It takes a single argument: the Target File Spec ID of the file you are interested in. It returns

TRUE if the file referenced by the File Spec exists and has a resource fork. CheckFileRsrcForkExists is a convenient way to see if a file which contains code exists..

CheckFileContainsRsrcByID

```
CheckFileContainsRsrcByID{ targetFileSpecRsrcID, resourceType, resourceID }
```

targetFileSpecRsrcID	A Target File Spec resource ID. (2-bytes)
resourceType	A resource type. (4-bytes)
resourceID	A resource ID. (2-bytes)

DESCRIPTION

Many times it is necessary to determine whether a specific file contains a specific resource. Use the checkFileContainsRsrcByID clause to determine whether a file has a resource with a known type and ID. checkFileContainsRsrcByID takes three arguments: the File Spec ID of the target file, the type of the resource, and the ID of the resource.

CheckFileContainsRsrcByName

```
CheckFileContainsRsrcByName{ targetFileSpecRsrcID, resourceType, resourceName }
```

targetFileSpecRsrcID	A Target File Spec resource ID. (2-bytes)
resourceType	A resource type. (4-bytes)
resourceName	A resource name. (even-padded Pascal string)

DESCRIPTION

Use checkFileContainsRsrcByName to determine whether a file has a resource with a known type and name. checkFileContainsRsrcByName takes three arguments: the File Spec ID of the target file, the type of the resource, and the name of the resource.

CheckFileVersion

```
CheckFileVersion{ targetFileSpecRsrcID, majorVersNum, minorVersNum, releaseStage,
releaseNum}
```

targetFileSpecRsrcID	A Target File Spec resource ID. (2-bytes)
majorVersNum	The major version number. (2-bytes)
minorVersNum	The minor version number. (2-bytes)
releaseStage	The release stage. Four release stage constants are defined in the InstallerTypes.r file: development, alpha, beta, final and release. (2-bytes)
releaseNum	The release number. (2-bytes)

DESCRIPTION

Use `CheckFileVersion` to check the version of a file. `CheckFileVersion` takes two arguments: the filespec ID of the file in question, and the minimal version of the file needed for this clause to be `TRUE`. Installer Engine obtains this information from the file's `'vers'` ID =1 resource. If no `'vers'` resource exists, `CheckFileVersion` returns `FALSE`.

For example, to check for System version 6.0.5 or newer, use the following clause.

```
checkFileVersion{SysFileSpec, 6, 5, release, 0};
```

To check for System version 7.1 or newer, use the following clause.

```
checkFileVersion{FinderFileSpec, 7, 0x10, release, 0};
```

To check for AppleTalk version 57.0.4 or newer, use the following clause.

```
checkFileVersion{ATalkFileSpec, 0x57, 0x04, release, 0};
```

CheckFileCountryCode

```
CheckFileCountryCode{ targetFileSpecRsrcID, countryCode }
```

`targetFileSpecRsrcID`	A Target File Spec resource ID. (2-bytes)
`countryCode`	A country code. (2 bytes)

DESCRIPTION

Use the `CheckFileCountryCode` clause to determine a file's country code (the name 'country code' has recently been changed to region code). `CheckFileCountryCode` takes two arguments: the File Spec ID of the file in question, and the country / region code required. Installer Engine obtains this information from the file's `'vers'` resource. The country (MPW 3.1) / region (MPW 3.2) codes can be found in the MPW interface file SysTypes.r.

CheckTgtVolSize

```
CheckTgtVolSize{ minimumSize, maximumSize }
```

`minimumSize`	The minimum size of the volume in Kbytes. (4-bytes)
`maximumSize`	The maximum size of the volume in Kbytes. (4-bytes)

DESCRIPTION

Use the `CheckTgtVolSize` clause to check target disk size. Note that this check is for disk size (to distinguish floppy from hard disks), not available disk space. Installer Engine itself handles the case where the target has insufficient space to accomplish the installation. `CheckTgtVolSize` takes two arguments: the smallest and largest size disks possible. To match any size disk, the minimal and maximal target disk sizes are both zero. If the minimum size is greater than zero but the maximum size equals zero, only the minimal requirement is used. Note that there are three size pairs defined in InstallerTypes.r. If you use {floppy, floppy} then checkTgtVolSize will return TRUE for 400 or 800k floppy targets. If you use {hdFloppy, hdFloppy} then checkTgtVolSize will return TRUE for high density floppy (1.4mb) targets. If you use {hardDisk, hardDisk} then checkTgtVolSize will return TRUE for any target volume of 10mb or greater.

NOTE

Since RAM disks may be most any size, make sure your rules handle this situation correctly. ◆

CheckRuleFunction

```
CheckRuleFunction{ ruleFunctionCodeRsrcID }
```

`ruleFunctionCodeRsrcID` The resource ID of a RuleFunction resource (‘inrf’). (2-bytes)

DESCRIPTION

Rule functions provide you with the ability to extend the rule-based decision-making scheme using a custom code resource written by you or someone else. When the `CheckRuleFunction` clause is evaluated the code resource is called and its return result determines whether the clause returns TRUE or FALSE.

To call a rule function code resource, first create a Rule Function resource (‘inrf’) that describes how to call the code resource. The Rule Function resource is described in the “InstallerTypes.r” file and the code resource can include the file “RuleFunctionHeader.h”.

When the code resource is called, the code resource receives the parameter block:

```
typedef struct {
    ProcPtr        fCallBackProcPtr;
    short              fTargetVRefNum;
    long               fTargetFolderDirID;
    short              fSystemVRefNum;
    long               fSystemBlessedDirID;
    long               fRefCon;
} RuleFunctionPBRec, *RuleFunctionPBPtr;
```

NOTE

The `fTargetFolderDirID` field will be -1 if Installer Engine is using disk mode, the folder does not exist, or a target File Spec referencing the reserved folder path `folder-user` has not been referenced by in another rule clause prior to calling the CheckRuleFunction rule clause. ◆

The code resource should use the function declaration:

```
long    RuleFunction( RuleFunctionPBPtr );
```

The function must return either a TRUE or FALSE value. Constants for these values are defined as:

```
#define kTRUERuleFunctionResult  1
#define kFALSERuleFunctionResult 0
```

▲ **WARNING**

If you choose to create a sub-heap for each invocation of the Rule Function please read the section “Running with a Sub-Heap” within the Atom Extender portion of this document. You can enter 0 (zero) in the requested memory field to not create a sub-heap and run inside Installer Engine’s heap. ▲

AddAssertion

```
AddAssertion{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to set.

DESCRIPTION

Use the AddAssertion clause to set a list of 1 or more assertions. AddAssertion takes a list of assertion constants as arguments. The constants are uniquely defined Rez symbols. addAssertion always returns TRUE.

CheckAllAssertions

```
CheckAllAssertions{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to check.

DESCRIPTION

Use CheckAllAssertions to check all of the constants a given list are currently asserted. CheckAllAssertions takes a list of assertion constants as arguments. checkAllAssertions returns TRUE if all of the items in the list have been asserted.

CheckAnyAssertion

```
CheckAnyAssertion{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to check.

DESCRIPTION

Use CheckAnyAssertion to check if one or more constants are currently asserted. CheckAnyAssertion takes a list of assertion constants as arguments. CheckAnyAssertion returns TRUE if one or more of the items in the list have been asserted.

CheckMoreThanOneAssertion

```
CheckMoreThanOneAssertion{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to check.

DESCRIPTION

CheckMoreThanOneAssertion is used to detect whether more than one constant in a list of constants are currently asserted. CheckMoreThanOneAssertion takes a list of assertion constants as arguments, and returns TRUE if more than one of the constants has been asserted. CheckMoreThanOneAssertion could be used, for example, to determine whether more than one printer is being installed. In that case, display a generic message reporting that “printer software” is being installed.

ClearAssertions

```
ClearAssertions{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to clear.

DESCRIPTION

Use the ClearAssertions clause to clear a list of one or more assertions. ClearAssertions takes a list of assertion constants as arguments. The constants are uniquely defined Rez symbols. ClearAssertions always returns TRUE.

CheckAllNonAssertions

```
CheckAllNonAssertions{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to check.

DESCRIPTION

Use CheckAllNonAssertions to check all of the constants a given list are currently not asserted. CheckAllNonAssertions takes a list of assertion constants as arguments. CheckAllNonAssertions returns TRUE if all of the items in the list are not currently asserted.

CheckAnyNonAssertion

```
CheckAnyNonAssertion{ { assertionValueList } }
```

assertionValueList	A list of 2-byte assertion values that you wish to check.

DESCRIPTION

Use CheckAnyNonAssertion to check if one or more constants are currently not asserted. CheckAnyNonAssertion takes a list of assertion constants as arguments. CheckAnyNonAssertion returns TRUE if one or more of the items in the list are currently not asserted.

AddAuditRec

```
AddAuditRec{ targetFileSpecRsrcID, selector, value }
```

targetFileSpecRsrcID	A Target File Spec resource ID. (2-bytes)
selector	The selector. (4-bytes)
value	The value (4-bytes)

DESCRIPTION

The AddAuditRec clause is used to add a record to a target file's audit record. AddAuditRec takes three arguments: a File Spec ID for the target file, an audit selector, and an audit value.

CheckAnyAuditRec

```
CheckAuditRec{ targetFileSpecRsrcID, auditSelector, {auditValueList} }
```

`targetFileSpecRsrcID`	A Target File Spec resource ID. (2-bytes)
`auditSelector`	The selector. (4-bytes)
`auditValueList`	A list of 4-byte values.

DESCRIPTION

The `CheckAnyAuditRec` clause is used to determine whether a given selector and value have been entered into the audit record of a target file (specified by the File Spec ID). `CheckAnyAuditRec` takes three arguments: a File Spec ID for the target file, an audit selector, and a list of audit values. If the value found in the audit resource for the specified selector is contained in the list of values the clause returns TRUE.

ReportVolError

```
ReportVolError{ errorString }
```

`errorString`	String to display. (Pascal String)

DESCRIPTION

`ReportVolError` is used to report an error because of a problem with the target disk (not a hard disk, disk size too small, no system, et cetera). The `ReportVolError` clause reports an error message in the Easy Install or Custom Install message area. `ReportVolError` always returns `TRUE`, so it can be called multiple times in a script. Messages are concatenated, just as with `addUserDescription`. Up to four lines can be displayed. `ReportVolError` takes a single even-padded Pascal string as an argument. To insert the name of the volume into the string, place the characters "`^0`" in the desired place.

If Installer Engine evaluates a `ReportVolError` clause, the Easy Install or Custom Install dialog displays a caution icon, and the Install button is dimmed. If you use this clause, the user can still switch volumes.

NOTE

Calling ReportVolError from within the global framework may not work correctly for some scripts. Depending on the rules within the easy or custom framework, the Install button may still be enabled. To solve this, call ReportVolError from within the easy or custom framework. ◆

ReportSysError

```
ReportSysError{ errorString }
```

`errorString`	String to display. (Pascal String)

DESCRIPTION

`ReportSysError` is used to report an error because of a problem with the system Installer Engine is running on (usually unsupported hardware or a fatal error). The `ReportSysError`

clause reports an error message to the Easy Install or Custom Install screen, which ever shows first. `ReportSysError` always returns `TRUE`, so it can be called multiple times in a script. Up to four lines can be displayed. Messages are concatenated, just as with `AddUserDescription`. `ReportSysError` takes a single even-padded Pascal string as an argument. If Installer Engine evaluates a `ReportSysError` clause, the Easy Install or Custom Install dialog displays a caution icon, and the Install button is dimmed. If you use this clause, the user cannot switch or eject volumes from the interface and has only one option: to quit.

Easy Feature Set Rule Clauses

AddPackages

```
AddPackages{ { assertionValueList } }
```

`packageIDList`	A list of 2-byte package resource (‘inpk’) IDs.

DESCRIPTION

The `AddPackages` clause is used to return a set of packages to the Easy Install screen. `AddPackages` can be called multiple times from a script, and the packages are unioned together. `AddPackages` takes a list of package IDs as arguments. `addPackages` always returns `TRUE`.

AddUserDescription

```
AddUserDescription{ messageString }
```

`messageString`	String to display. (Pascal String)

DESCRIPTION

Use `AddUserDescription` to add to the text which is shown in the Easy Install dialog. `AddUserDescription` takes a single even-padded Pascal string as an argument. The text to be added is appended to whatever text has been previously added using the `AddUserDescription` clause. Up to four lines can be displayed. `AddUserDescription` always returns `TRUE`. Note that Installer Engine will not supply introductory information so you need to say something like: “`Click the Install button to install\n`{your product here}.” Use the "`\n`" characters to insert a return character into the description string.

Custom Install Rule Clauses

AddCustomItems

```
AddCustomItems{ { integer packageRsrcID } }
```

`packageRsrcID`	A list of 2-byte resource IDs of ‘inpk’ resources to append to the top level of the Custom Install list.

DESCRIPTION

The `AddCustomItems` rule clause appends the specified packages to the top level of the custom install list regardless of the setting of its `showsOnCustom` flag. Each package is appended in the order listed. All sub-packages with the `showsOnCustom` flag set are appended at the appropriate sub-level.

Using the Installer Preference Resource ('inpr')

The preference resource allows the scriptwriter to override certain default interface features and actions of Installer Engine. To utilize this control you must add a preference resource ('inpr' ID=300) to the script file. Installer Engine contains a default 'inpr' resource with the ID of 305. Installer Engine first looks for your 'inpr' with the ID 300, and if not found uses its default resource.

The fields and flags in the preference resource control two main areas of Installer Engine, the recommended target disk / folder selection, and specific human interface elements.

Using the Target Disk Interface

If your product must always be installed into a specific location on the target disk, then you'll probably want to limit the user to only selecting from the mounted volumes. This is the case for products, such as system software, that need to be installed predominately in the System Folder. Setting the `useDiskTargetMode` flag in the preference resource tells Installer Engine to present an interface compatible with the old Installer 3.X.

Using the Application Folder Interface

If your script installs a product such as an application which can be placed most anywhere on the user's Macintosh, then you'll probably want to allow the user to select a target folder.

Much like the current `special-xxxx` folder path identifier, the new `folder-user` folder path identifier will be the folder path selected by the user. When the user clicks the Select Folder button a modified standard file dialog appears to allow selection of a new or existing folder. This dialog is discussed in the section "Using the Select Folder Dialog".

Note: the target selection plug-in in Upgrader does not currently allow selection of an application folder.

Installer Preference Resource Reference

This section describes the resource description of the 'inpr' resource.

Resource Description

Format 1 of the 'inpr' resource:

```
#define preferenceFlags1                                    \
    boolean useDiskTargetMode, useFolderTargetMode;         \
    fill bit[2];                                            \
    boolean noSetupFunctionSupplied, setupFunctionSupplied; \
    boolean dontAllowCleanInstall, allowCleanInstall;       \
    fill bit[1];                                            \
    boolean    isNotSSWInstallation, isSSWInstallation;     \
    fill bit[9];
```

```
#define featureSetFlags                                     \
       fill bit[16];


type 'inpr' {
       switch {
           case format1:
               key integer = 1;      /* Preference version */
               preferenceFlags1;     /* Preference Flags */
               rsrcType;             /* Type of code resource for Setup Function */
               rsrcID;               /* Id of code resource for Setup Function */
               longint;              /* Text Encoding ID of product's language */
               rsrcID;               /* ID of 'STR#' rsrc with feature set names */
               unsigned integer = $$CountOf(featureSetList);  /* Feature Set List */
               wide array featureSetList {
                   featureSetFlags;  /* Feature set flags */
                   rsrcID;           /* Feature set rule framework */
                   integer;          /* String index of feature set name */
               };
               evenPaddedString;     /* Default Folder name for folder mode. */

       };
};
```

Flag descriptions

`useDiskTargetMode/useFolderTargetMode`
: Specifies whether the user can choose a target folder, or is limited to choosing a target disk. If the `useDiskTargetMode` flag is specified, “Switch Disk ” and “Eject Disk” buttons are shown to the user. This is very similar to the old Installer 3.X. If the ‘`useFolderTargetMode`’ flag is specified, the interface appears with a “Select Folder…” button, instead of the “Switch Disk ” and “Eject Disk” buttons.

`noSetupFunctionSupplied/setupFunctionSupplied`
: Specifies whether Installer Engine should call the setup function. The type and ID of the setup function code resource must be entered in the setup function fields when the `setupFunctionSupplied` flag is specified.

`dontAllowCleanInstall/allowCleanInstall`
: Specifies whether Installer Engine should allow a clean a install of this software component. **WARNING**: Only Installer scripts that install an entire System Folder should allow a clean installation. Generally, this flag should only be used by Apple Computer, Inc.

`isNotSSWInstallation/isSSWInstallation`
: Specifies whether this software component installs an entire System Folder. **WARNING**: Generally, this flag should only be used by Apple Computer, Inc.

Field descriptions

`Setup Function Code Resource Type`
: The resource type of the setup function code resource. This field is ignored unless the `setupFunctionSupplied` flag is specified. The type is usually ‘infn’. (4-bytes)

`Setup Function Code Resource ID`
: The resource ID of the setup function code resource. This field is ignored unless the `setupFunctionSupplied` flag is specified. (2-bytes)

Text Encoding ID
: Enter the text encoding value that corresponds to the localized region of the software component you are installing. Entering the correct value helps conversion of Unicode-based file names when installing onto Extended format volumes. (4-bytes)

Feature Names ‘STR#’ ID
: The ID of a ‘STR#’ resource containing the names of the features sets listed below. (2-bytes)

Feature Set Framework ID
: The ID of a ‘infr’ resource defining the feature set. (2-bytes)

Feature Name Index
: The index of the string in the ‘STR#’ resource specified above that contains the name of the feature set to be displayed to the user. (2-bytes)

`Recommended Target Folder Name`
: The recommended target folder name. This can be specified at run-time by the scriptwriter in the setup function. As with any file name, the length should be limited to 31 characters, although Installer Engine will truncate the string if too long. (even-padded Pascal string)

Data Structures

The setup function is a code resource specified in the preference resource that is passed a pointer to a parameter block with information about Installer Engine’s environment. The entry point of this code resource must have the interface:

```
OSErr  EnvironmentSetupFunction( EnvironmentSetupPBPtr );
```

Return the result code noErr to have Installer Engine continue as normal. Return the result code `kQuitInstallerNow` to force Installer Engine to quit immediately. Returning any other value will cause an error alert to be display before forcing Installer Engine to quit.

The parameter block contains Installer Engine’s suggested target application folder and system disk. The fields in the parameter block can be changed to override the suggested values.

```
typedef  struct {
 —> ProcPtr       fCallBackProcPtr;
<—> FSSpec        fTargetFSSpec;
<—> short         fSystemVRefNum;
} EnvironmentSetupPB, *EnvironmentSetupPBPtr;
```

Field descriptions

`fCallBackProcPtr`
: A pointer to Installer Engine’s dispatch routine. You’ll need to pass this field as a parameter to Installer function glue routines.

`fTargetFSSpec`
: Contains Installer Engine’s suggested target folder information. You can choose to override this suggestion by changing any of the fields in the `FSSPec` structure. If the `useDiskTargetMode` flag is specified in the preference

	resource, only the `vRefNum` field of `fTargetFSSpec` is used by Installer Engine.
`fSystemVRefNum`	The `fSystemVRefNum` field contains Installer Engine's recommended System Folder volume. You can choose to override this suggestion by changing the value passed. This field is ignored when using the `useDiskTargetMode` flag because the system disk must always be the same as the target disk.

CHAPTER 3

Defining Actions

Actions you wish to perform during an installation or removal are specified using atoms. There are eight different types of atoms, each performing a unique type of action, such as installing a file or executing your own custom code resource. In order to have your actions performed, you must reference them by ID from the package resources you defined for your user interface.

Atoms available:

File Atom	Copies or deletes one or both forks of a Macintosh file.
Resource Atom	Copies or deletes a single resource inside a file.
Font Atom	Copies or deletes a font family inside a file. NOTE: Apple recommends installing fonts as entire suitcase files instead of using the font atom.
ResMerge Atom	Merges all resources of a source file into the target file.
Folder Atom	Copies all files from a source folder into a target folder.
Action Atom	Executes a custom code resource at the beginning or at the end of the installation .
Boot Block Atom	Updates the book blocks of a target volume. NOTE: This atom is only used for installation of system software.
Audit Trial Atom	Updates information stored in an audit trail resource in the target file.

Before we delve into the details of each atom, let's continue our example we started in the previous chapter.

Example (Part 2): Defining the Actions

You might have noticed in part 1 of this example, the 'inpk resources we defined already contained references to file atoms, which we now need to create.

Step 8: Create the File Atom Resources

Since this example Installer script is pretty simple, we'll use the same IDs for our file atoms as the package resources. For the seven features selectable by the user, we'll create seven file atoms using the new 'ifa#' resource type:

```
resource 'ifa#' (300) {
    format0 {
        {   /* [1] */  1001,                // Unique ID of this File Atom
                       deleteWhenRemoving,  // Delete file on removal
                       deleteWhenInstalling, // Doesn't matter since we're copying
                       copy,                // Copy file during installation
                       dontIgnoreLockedFile, // Warn user if file is locked
                       dontSetFileLocked,   // Leave file unlocked after installation
                       useVersProcToCompare, // Compare newness of file based on 'vers' resource
                       srcNeedExist,        // File must exist on source disk
                       rsrcForkInRsrcFork,  // File isn't compressed so rsrc fork is in rsrc fork
                       leaveAloneIfNewer,   // Don't update an existing newer file
                       updateExisting,      // Go ahead an update if appropriate
                       copyIfNewOrUpdate,   // Go ahead and create a new file when necessary
                       rsrcFork,            // Copy the resource fork, if any
                       dataFork,            // Copy the data fork, if any
                       0,                   // Total file size, filled in by ScriptCheck
                       0,                   // Finder attributes, filled in by ScriptCheck
                       1001,                // Reference to target file specification
                       1001,                // Reference to source file specification
                       0,                   // Exact target data fork size, filled in by ScriptCheck
                       0,                   // Exact target rsrc fork size, filled in by ScriptCheck
                       0,                   // Source version number, filled in by ScriptCheck
                       0,                   // Compare versions using built-in compare proc.
                       0,                   // Zero because we're not copying compressed data
        /* [2] */  1002, deleteWhenRemoving, deleteWhenInstalling, copy, dontIgnoreLockedFile,
                       dontSetFileLocked, useVersProcToCompare, srcNeedExist, rsrcForkInRsrcFork,
                       leaveAloneIfNewer, updateExisting, copyIfNewOrUpdate, rsrcFork, dataFork,
                       0, 0, 1002, 1002, 0, 0, 0, 0, 0,
        /* [3] */  2001, deleteWhenRemoving, deleteWhenInstalling, copy, dontIgnoreLockedFile,
                        dontSetFileLocked, useVersProcToCompare, srcNeedExist, rsrcForkInRsrcFork,
                        leaveAloneIfNewer, updateExisting, copyIfNewOrUpdate, rsrcFork, dataFork,
                       0, 0, 2001, 2001, 0, 0, 0, 0, 0,
        /* [4] */  3001, deleteWhenRemoving, deleteWhenInstalling, copy, dontIgnoreLockedFile,
                       dontSetFileLocked, useVersProcToCompare, srcNeedExist, rsrcForkInRsrcFork,
                       leaveAloneIfNewer, updateExisting, copyIfNewOrUpdate, rsrcFork, dataFork,
                       0, 0, 3001, 3001, 0, 0, 0, 0, 0,
        /* [5] */  3002, deleteWhenRemoving, deleteWhenInstalling, copy, dontIgnoreLockedFile,
                       dontSetFileLocked, useVersProcToCompare, srcNeedExist, rsrcForkInRsrcFork,
                       leaveAloneIfNewer, updateExisting, copyIfNewOrUpdate, rsrcFork, dataFork,
                       0, 0, 3002, 3002, 0, 0, 0, 0, 0,
        /* [6] */  1004, deleteWhenRemoving, deleteWhenInstalling, copy, dontIgnoreLockedFile,
                       dontSetFileLocked, useVersProcToCompare, srcNeedExist, rsrcForkInRsrcFork,
                       leaveAloneIfNewer, updateExisting, copyIfNewOrUpdate, rsrcFork, dataFork,
                       0, 0, 1004, 1004, 0, 0, 0, 0, 0,
        /* [7] */  1005, deleteWhenRemoving, deleteWhenInstalling, copy, dontIgnoreLockedFile,
                       dontSetFileLocked, useVersProcToCompare, srcNeedExist, rsrcForkInRsrcFork,
                       leaveAloneIfNewer, updateExisting, copyIfNewOrUpdate, rsrcFork, dataFork,
                       0, 0, 1005, 1005, 0, 0, 0, 0, 0,
        }
    }
};
```

In the next chapter the example we will continue with the definition of the source and target file specifications.

Using the File Atom (‘infa’ & ‘ifa#’)

The File Atom can be used to copy or delete one or both forks of a file. You have a choice of two resources to describe the file action. The older ‘infa’ resource allows files to be split across multiple source disk, but the new ‘ifa#’ resource is more efficient and easier to use and is our recommended choice. You can mix both resource types as long as you ensure each file atom has a unique ID.

The File Atom provides the following major features for copying or deleting files:

- Ability to update a file only if it already exists.
- Ability to preserve an existing file.
- Ability to preserve a newer file based on its creation date, ‘vers’ 1 resource or custom code you write.
- Decompression of a file during installation.
- Automatically unlock the file before replacing or deleting, and/or lock the file when the installation is finished.
- Install a file that has been split among multiple source files (not supported when using the ‘ifa#’ resource).

If you are simply copying hundreds of uncompressed files and do not wish to create separate Files Atoms for each, you might be able to use a Folder Atom.

Storing File Atoms in the ‘ifa# Resource

When referencing a File Atom from a package resource, you must always use the ‘infa’ atom type, regardless if it is stored as a ‘infa’ resource or in a ‘ifa#’ resource. When looking for a file atom by ID, Installer Engine will look first for a file atom with the specified record ID in the ‘ifa#’ resource, and if not found will look for a ‘infa’ resource with the specified ID. NOTE: an Installer script can only have one ‘ifa#’ resource.

Comparing Files by Version

Two new fields in the File Atom allow the scriptwriter to compare the source and target file using their version number, instead of only using their creation date. The first field holds the source file’s version number, and the second field holds the resource ID of a new Version Compare (‘invc’) script resource. The Version Compare script resource allows the scriptwriter to call a code resource that calculates the version number of the target file.

To compare the newness of the target and source using the old creation date method, use the `useSrcCrDateToCompare` flag. To compare using the version number in the target’s ‘vers’ ID=1 resource then use the `useVersProcToCompare` flag and place a 0 (zero) in the Version Compare Rsrc ID field. If the version number is stored somewhere besides the ‘vers’ ID=1 resource then you’ll need to create and attach a Version Compare script resource to the File Atom.

Using either the `useSrcCrDateToCompare` or `useVersProcToCompare` flag presents the identical interface to the user if the source file is older than the target file. See the description of the `leaveAloneIfNewer` flag in the “File Atom Reference” section.

Using Split Sources with File Atoms

The new ‘infa’ script resource contains a source list that holds zero or more references to source files. If the original source file must be split into smaller files, an entry for each split source piece should be placed into the source list. Each entry contains information about the location of the source file (‘infs’ ID), the target size of the resource fork piece and the target size of the data fork piece.

Each target size field must contain the exact size the source piece will appear in the target file. If the piece is being decompressed with an Atom Extender during installation, then this field must contain the uncompressed size of the piece. If only one fork is requested to be copied, the target size field of the other fork is ignored. Place a zero in a fork’s target size field if the fork contains no data.

NOTE

The order of the entries in the source list is important. Each piece will be written to the target file in the same order as it appears in the list. ◆

Using Atom Extenders with File Atoms

Using Atom Extenders with the new File Atom (Format 1) allows scriptwriters to easily decompress files that have been compressed on the source installation disk. Attach the Atom Extender to the File Atom file by entering the ID of the ‘inex’ script resource into the appropriate field in the ‘infa’. The code resource referenced from the ‘inex’ script resource will be called at the desired point in the installation so it can read the compressed file data, decompress the data, then write it to the target file.

NOTE

You should never create a source file that has a compressed resource fork, because any attempt to access the resource fork by the Finder or other application may crash the Macintosh. The `rsrcForkInRsrcFork/rsrcForkInDataFork` flag has been added to designate where the resource fork is stored. Use the `rsrcForkInDataFork` flag when copying a compressed resource fork that is stored in the data fork of the source file. Use the `rsrcForkInRsrcFork` flag when copying a non-compressed resource fork. ◆

When Installer Engine is preparing to install, each File Atom is expanded into one or more parts. As each part is installed, the Atom Extender attached to the original File Atom will be called for each part. When only one source file is specified there is one part for each fork being copied. When split source pieces are specified there is one part for each fork being copied from each piece.

The goal of the new File Atom and Atom Extender is to provide compatibility with most popular compressed source file configurations. Several configurations are listed below.

- The simplest source file configuration is for each compressed fork to be placed into the data fork of two separate files. Installer Engine will call the Atom Extender separately for each fork.
- Another source file configuration places both forks into the same data fork of a source file, with a header describing the format of the data. The Atom Extender will be called once for each fork copied, but will be responsible for finding, reading and writing the correct data for the specified fork. The Atom Extender parameter block will provide information about which fork is being copied.
- An “archived” source file configuration contains multiple compressed files (and forks) in one source file. One File Atom must be created for each target file being copied, with

each referring to the same source file (‘infs’). The Atom Extender parameter block will contain information about the current file (and fork) being copied, with which the Atom Extender will find, read, decompress and write the proper target file’s data. Our InstaCompOne Atom Extender is an example of this approach.

For those source configurations that cannot be accommodated by the built-in routines, scriptwriters can use the supplied parameter block to perform the copy by themselves. This strategy should only be taken when absolutely necessary.

Installing a Custom Folder Icon

The File Atom can easily be used to set the custom icon of any folder on the target disk. Installer Engine notices if the scriptwriter is copying the special invisible “Icon\n” file, and if so automatically sets the userCustomIcon bit on its parent folder.

There are several key points to remember when installing a custom icon:

- Since Installer Engine does not remove the special “Icon\n” file correctly, use the `dontDeleteWhenRemoving` flag to prevent its removal.
- It’s polite to preserve an existing custom icon, so use the `keepExisting` flag.
- Installer Engine will install the custom icon even if the directory is the root level of the hard disk. We discourage scriptwriters from recommending the root level of the hard disk as the target folder, but user’s can select it if they choose.

File Atom Reference

This section describes the resource description of the ‘infa’ and ‘ifa#’ resources.

Resource Descriptions

File Atom Resource (‘infa’)

```
#define fileAtomFlags                                                   \
        boolean    dontDeleteWhenRemoving, deleteWhenRemoving;          \
        boolean    dontDeleteWhenInstalling, deleteWhenInstalling;      \
        boolean    dontCopy, copy;                                      \
        fill bit[3]    /* Reserved */                                   \
        boolean    dontIgnoreLockedFile, ignoreLockedFile;              \
        boolean    dontSetFileLocked, setFileLocked;                    \
        boolean    useSrcCrDateToCompare, useVersProcToCompare;         \
        boolean    srcNeedExist, srcNeedNotExist;                       \
        boolean    rsrcForkInRsrcFork, rsrcForkInDataFork;              \
        boolean    updateEvenIfNewer, leaveAloneIfNewer;                \
        boolean    updateExisting, keepExisting;                        \
        boolean    copyIfNewOrUpdate, copyIfUpdate;                     \
        boolean    noRsrcFork, rsrcFork;                                \
        boolean    noDataFork, dataFork


type 'infa' {
        switch {
            case format1:
                key integer = 1;                /* File Atom version */
                fileAtom1Flags;                 /* File Atom Flags */
```

```
            unsigned longInt;               /* Total Target File Size */
            unsigned integer;               /* Finder Attribute Flags */
            fileSpecID;                     /* Tgt file spec ID */
            integer = $$CountOf (Pieces);   /* Number of Source Pieces */
            wide array Pieces {
                fileSpecID;                     /* Source File Spec*/
                unsigned longInt;               /* Target Data Fork Part Size */
                unsigned longInt;               /* Target Rsrc Fork Part Size */
            };
            unsigned longint;               /* Source Version Number */
            rsrcID;                         /* Version Compare Rsrc ID */
            rsrcID;                         /* Atom Extender Rsrc ID */
            evenPaddedString;               /* Atom Description */
        };
};
```

Flag Descriptions

`dontDeleteWhenRemoving/deleteWhenRemoving`
: Determines if the file is deleted during a removal. When using the `deleteWhenRemoving` flag, if the target file exists and the user clicks Remove, the target file is deleted. If the file does not exist on the target disk, this flag is ignored. The file atom must be part of a package that uses the `removable` flag. Note that of the following File Atom flags, only the `rsrcFork/noRsrcFork` and `dataFork/noDataFork` flags have any effect on the removal process.

`dontDeleteWhenInstalling/deleteWhenInstalling`
: Determines if a file is deleted during an installation when using the `dontCopy` flag. Installer generally ignores this flag when using the `copy` flag. When using the `deleteWhenInstalling` flag, after the user clicks Install, the target file is deleted if it exists, otherwise the flag is ignored. The `deleteWhenInstalling` flag is primarily used for deleting previously installed files that are no longer needed.

`dontCopy/copy`
: Determines if the file is copied during an installation. Note that some flags (`leaveAloneIfNewer`, `keepExisting`, `copyIfUpdate`) can prevent copying from happening under the circumstances specified.

`dontIgnoreLockedFile/ignoreLockedFile`
: Determines if an existing locked target file should be automatically unlocked before replacing. Only use the `ignoreLockedFile` flag if you originally installed the file locked or your software locks the file. Use the `dontIgnoreLockedFile` flag the majority of times to preserve the user's control over their Macintosh.

`dontSetFileLocked/setFileLocked`
: Determines if the target file should be locked after copying. Use `setFileLocked` to request that the target file be locked at the end of the installation.

`useSrcCrDateToCompare/useVersProcToCompare`
: Determines how Installer Engine will determine if the

target file is newer or older than the source file. If using the `useSrcCrDateToCompare` flag the creation date entered in the Source File Spec is compared with the creation date of the target file. If the `useVersProcToCompare` flag is used, an optionally supplied version function is called to determine the version number of the target file. See the description of the `Version Compare Rsrc ID` field.

`srcNeedExist/srcNeedNotExist`

Determines whether the source file must exist on the source disk. Use `srcNeedNotExist` if the source file can optionally reside on the source disk. If the file is not found the atom is ignored and the installation continues.

`rsrcForkInRsrcFork/rsrcForkInDataFork`

Since a compressed resource fork should never be left in the resource fork of the source file, this allows the scriptwriter to easily store the compressed resource fork in the data fork. These flags determine where the resource fork data will be read from when using the ReadSourceData() routine. If the `rsrcForkInDataFork` flag is specified, the source data will actually be read from the data fork.

`updateEvenIfNewer/leaveAloneIfNewer`

Determines what action Installer Engine should take if the target file is newer than the source file that is replacing it. The method Installer Engine uses to determine the newness of the files is based on the `useSrcCrDateToCompare` and `useVersProcToCompare` flags. Use the `updateEvenIfNewer` flag if the version of this file must be synchronized with specific versions of other files that are part of the installation. The alert displayed to the user depends whether the user is performing an Easy Install or a Custom Install.

Easy Install — If using the `leaveAloneIfNewer` flag, the user will not be notified and the newer will be preserved. If using the `updateEvenIfNewer` flag, an alert is shown that has two options: Continue or Cancel. Clicking Cancel will stop the installation, and clicking Continue will replace the newer target file with the older source file.

Custom Install — If using the `leaveAloneIfNewer` flag, the user will be presented with an alert that provides three choices: Newer, Older or Cancel. If using the `updateEvenIfNewer` flag, the alert is the same as for an Easy Install.

`updateExisting/keepExisting`

Allows the scriptwriter to preserve an existing file when using the `copy` and/or `deleteOnInstall` flag. This might be the case if you want to preserve a preference file that contains user specific data. Use `keepExisting` to prevent Installer Engine from disturbing an existing target file. No copying will occur if the `keepExisting` flag is used with the `copyIfUpdate` flag.

`copyIfNewOrUpdate/copyIfUpdate`
: Allows the scriptwriter to update a file only if it already exists when used with the `copy` flag. Use `copyIfUpdate` to prevent a new file from being created. If the `copyIfUpdate` flag is used with the `keepExisting` flag then no copying will occur.

`noRsrcFork/rsrcFork`
: Determines if the resource fork of the file is affected during an installation or removal. Use `resourceFork` to copy or delete the entire resource fork of the file during an installation, or delete the entire resource fork during a removal. Use `noResourceFork` to not touch the resource fork of the file.

`noDataFork/dataFork`
: Determines if the data fork of the file is affected during an installation or removal. Use `dataFork` to copy or delete the entire data fork of the file during an installation, or delete the entire data fork during a removal. Use `noDataFork` to not touch the data fork of the file.

Field Descriptions

`Total Target File Size`
: The size in bytes of the file to be installed or deleted. This field is only used by Installer Engine in figuring the disk size needed for an installation. (4-bytes)

`Finder Attribute Flags`
: These flags determine how Finder displays and manages user interaction with the file. If you are using ScriptCheck 4.0 or later, then this field is updated automatically (see ScriptCheck documentation for handling compressed source files), otherwise you must place the correct value in this field. See *Inside Macintoh, Volume VI*, pages 9-36 - 9-38 for more information about these flags. (2-bytes)

`Source File Spec. Rsrc ID`
: The resource ID of a Source File Spec. (‘infs’) describing the source file where the data is stored. If you’re just deleting a file and therefore don’t need a source file, enter 0 (zero) in this field. (2-bytes)

`Target Data Fork Piece Size`
: The number of bytes the data fork piece will occupy in the target file. If the data fork is decompressed using an Atom Extender during the installation, then this value should be the original noncompressed size of the source piece. The value is used to compute where multiple pieces are to be written into the final target file, and therefore must be exact. If not copying the data fork, this field is ignored. (4-bytes)

`Target Rsrc Fork Piece Size`
: The number of bytes the resource fork piece will occupy in the target file. If not copying the resource fork, this field is ignored. If the resource fork is decompressed using an Atom Extender during the installation, then this value should be the original noncompressed size of the source piece. The value is used to compute where multiple pieces are to be

written into the final target file, and therefore must be exact. (4-bytes)

`Source Version Number` The version number (BCD format) of the source file when using the `useVersProcToCompare` flag. See *Inside Macintosh: Macintosh Toolbox Essentials*, page 7-31 for a description of the version number format. (4-bytes)

`Version Compare Rsrc ID` The resource ID of a Version Compare ('invc') script resource. The resulting version number from the version function will be compared with the `Version Number` field to determine if the target is newer or older than the source. If the resource ID is 0 (zero) and the `useVersProcToCompare` flag is used Installer Engine will default to comparing the `Source Version Number` field with the version number in the 'vers' ID=1 resource contained in the target file. If no 'vers' ID=1 resource exists then the target version number is assumed to be 0 (zero). (2-bytes)

`Atom Extender Rsrc ID` The resource ID of an Atom Extender ('inex' script resource). (2-bytes)

`File Atom Description` The File Atom Description field is an even-padded Pascal string describing the atom. This is used as part of the status dialog. If you do not supply a description Installer Engine will display "Copying File: [file name]" or "Writing File: [file name]." If you supply a description it will be appended to the string "Copying " or "Writing ".

File Atom List Resource ('ifa#')

```
type 'ifa#' {
       switch {
           case format0:
               key integer = 0;                    /* Data Format version */
               integer = $$CountOf (FileAtomRec);  /* Number of records */
               wide array FileAtomRec {
                   longint;                        /* Record ID */
                   fileAtom1Flags;                 /* File Atom Flags */
                   unsigned longInt;               /* Total Target File Size */
                   unsigned integer;               /* Finder Attribute Flags */
                   fileSpecID;                     /* Tgt file spec ID */
                   fileSpecID;                     /* Source File Spec*/
                   unsigned longInt;               /* Target Data Fork Part Size */
                   unsigned longInt;               /* Target Rsrc Fork Part Size */
                   unsigned hex longint;           /* Source Version Number*/
                   rsrcID;                         /* Version Compare Rsrc ID */
                   rsrcID;                         /* Atom Extender Rsrc ID */
               };
       };
};
```

Flag Descriptions

See flag descriptions of 'infa' resource for information about flags

Field Descriptions

`Record ID` An ID that uniquely identifies this file atom. When combining multiple 'infa' and 'inf#' resource within the

same Installer script, make sure that no ID of an ‘infa’ resource is used to identify a file atom record with a ‘ifa#’ resource. (4-bytes)

See field descriptions of ‘infa’ resource for information on the other fields

Using the Resource Atom

The Resource Atom should be used to copy or delete individual resources in a file. Like the file atom, you have a choice of two resources to describe the resource action. The older ‘inra’ resource allows resources to be split across multiple source disk, but the new ‘inr#’ resource is more efficient and easier to use and is our recommended choice. You can mix both resource types as long as you ensure each resource atom has a unique ID.

The resource atom provides the following major features for copying or deleting resources:

- Ability to update a resource only if it already exists.
- Ability to preserve an existing resource.
- Ability to preserve a newer resource based on its version number that is determined by custom code you write.
- Decompression of the resource during installation.
- Installation of a resource that has been split among multiple resources on multiple source disks (not supported when using the ‘inr#’ resource).

If you need to copy all the resources from a single source file and merge them with the target file you may want to consider using the ResMerge Atom.

Storing Resource Atoms in the ‘inr# Resource

When referencing a Resource Atom from a package resource, you must always use the ‘inra’ atom type, regardless if it is stored as a ‘inra’ resource or in a ‘inr#’ resource. When looking for a Resource Atom by ID, Installer Engine will look first for a Resource Atom with the specified record ID in the ‘inr#’ resource, and if not found will look for a ‘inra’ resource with the specified ID. NOTE: an Installer script can only have one ‘inr#’ resource.

Use the ‘infa’ resource for those Resource Atoms that need any of the following features:

- Find a source resource based on its name.
- Copy a resource that is split across multiple files.
- Specify a target resource name.
- Use with InstaCompOne decompression Atom Extender.
- Change the type or ID during installation. The Resource List Atom requires the target resource type and ID to be the same as the source resource type and ID.

Comparing Resources By Version

Two new fields in the Resource Atom allow the scriptwriter to compare the source and target resource based on the version number. The first field holds the source file’s version number, and

the second field holds the ID of a Version Compare ('invc') script resource. The Version Compare script resource allows the scriptwriter to call a code resource that calculates the version number of the target resource.

To compare the version number of the target resource to the source version number entered in the Resource Atom you'll need to create and attach a Version Compare script resource to the Resource Atom. Since setting and obtaining version number information for an individual resource is not a feature of the Resource Manager, extracting the version number from the target resource depends on how the resource encodes its version number. Most examples of this use a header at the beginning of the resource to hold this type of information.

Using Split Sources with Resource Atoms

The format 1 version of the Resource Atom script resource contains a list of source entries. If the original resource must be split into smaller resource pieces, an entry for each split source piece must be placed into the source list. Each entry contains information about the location of the source file ('infs' ID), the source resource type, the source resource ID, the target size of the resource piece, and the source resource name. The target size field must contain the exact size the source piece will appear in the target resource. If the piece is being decompressed with an Atom Extender during installation, this field must contain the original uncompressed size of the piece.

When specifying more than one split resource piece, Installer Engine joins the pieces into a target resource in the order the sources are specified. Generally, you'll want to use 'part' for the type of source resources that have been split and/or compressed. This prevents resources compressed using third-party schemes from being confused with the original resource, especially when using tools like ResEdit and DeRez.

Using Atom Extenders with Resource Atoms

You'll use an Atom Extender to decompress a resource that was compressed by you to save space on your source disks. The Resource Atom is expanded into one or more parts that the Atom Extender receives with the `kBeforePart` and `kAfterPart` messages. There will be one part for each source resource piece specified in the source list. Since owned resources are always copied whole and are usually relatively small, the owner's Atom Extender is never called when copying its owned resources.

NOTE

The `compressed` flag in the resource attributes should NOT be set on any source resource being decompressed using an Atom Extender. This flag is presently reserved for use by Apple Computer, Inc. ◆

Resource Atom Reference

This section describes the resource description of the 'inra' and 'inr# resources.

Resource Description

Resource Atom Resource ('inra')

```
#define resourceAtomFlagsFormat1                                    \
```

```
        boolean    dontDeleteWhenRemoving, deleteWhenRemoving;        \
        boolean    dontDeleteWhenInstalling, deleteWhenInstalling;    \
        boolean    dontCopy, copy;                                    \
        fill bit[5];                          /* Reserved    */      \
        boolean    updateEvenIfNewer, leaveAloneIfNewer;              \
        boolean    noTgtRequired, tgtRequired;                        \
        boolean    updateExisting, keepExisting;                      \
        boolean    copyIfNewOrUpdate, copyIfUpdate;                   \
        boolean    dontIgnoreProtection, ignoreProtection;            \
        boolean    srcNeedExist, srcNeedNotExist;                     \
        boolean    byName, byID;                                      \
        boolean    nameNeedNotMatch, nameMustMatch

type 'inra' {
        switch {
            case format1:
                key integer = 1;             /* Resource Atom Format vers */
                resourceAtomFlagsFormat1;    /* Resource Atom Flags */
                unsigned longInt;            /* Total Target Size  */
                fileSpecID;                  /* Target File Spec */
                rsrcType;                    /* Target Resource Type */
                rsrcID;                      /* Target Resource ID */
                integer;                     /* Target Resource Attributes */
                evenPaddedString;            /* Target Resource Name */
                integer = $$CountOf (Parts); /* Number of Pieces */
                wide array Parts {
                    fileSpecID;                 /* Source Piece File Spec */
                    rsrcType;                   /* Source Piece Type  */
                    rsrcID;                     /* Source Piece Rsrc ID */
                    unsigned longInt;           /* Target Piece Size  */
                    evenPaddedString            /* Source Piece Rsrc Name */
                }
                unsigned longint;            /* Source Version Number */
                rsrcID;                      /* Version Compare Rsrc ID */
                rsrcID;                      /* Atom Extender Rsrc ID */
                evenPaddedString;            /* Atom Description */
        };
};
```

Flag Descriptions

`dontDeleteWhenRemoving/deleteWhenRemoving`

Determines if the resource is deleted during a removal. When using the `deleteWhenRemoving` flag, if the target resource exists and the user clicks Remove, the target resource is deleted. If the resource does not exist on the target disk, this flag is ignored. Note that the Resource Atom must be part of a package that uses the `removable` flag.

`dontDeleteWhenInstalling/deleteWhenInstalling`

Determines if the resource is deleted during an installation when using the `dontCopy` flag. When using the `deleteWhenInstalling` flag, after the user clicks Install, the target resource is deleted if it exists, otherwise the flag is ignored. The `deleteWhenInstalling` flag is primarily used for deleting previously installed resources that are no longer needed. Note that some flags (`leaveAloneIfNewer`, `keepExisting`) can prevent deletion from happening under the circumstances specified.

`dontCopy/copy` Determines if the resource is copied or not during an installation. Note that some flags (`leaveAloneIfNewer`, `keepExisting`, `copyIfUpdate`) can prevent copying from happening under the circumstances specified.

`updateEvenIfNewer/leaveAloneIfNewer`
Determines what action Installer Engine should take if the target resource is newer than the source resource that is replacing it. The scriptwriter must write a code resource that determines the version number by referencing the resource ID of a Version Compare ('invc') script resource. Use the leaveAloneIfNewer flag to prevent the older source resource from replacing the newer target resource during an installation. Unlike the File Atom, the user is not given a chance to override the result of this flag. Use the updateEvenIfNewer flag if the version of this resource must be synchronized with specific versions of other files or resources that are part of the installation.

`noTgtRequired/tgtRequired`
The target file for this resource must already exist. If the target file does not exist, the user is warned that a target file is needed. (For example, AppleShare installs resources into the System file, but if there is no system file, Installer Engine alerts the user, rather than creating a System file which has only the AppleShare resources in it.) When using the `noTgtRequired` flag and the target file does not exist, Installer Engine will create one.

`updateExisting/keepExisting`
Allows the scriptwriter to preserve an existing resource when using the `copy` and/or `deleteOnInstall` flag. This might be the case if you want to preserve a preference resource that contains user specific data. Use `keepExisting` to prevent Installer Engine from disturbing an existing target resource. No copying will occur if the `updateExisting` flag is used with the `copyIfUpdate` flag.

`copyIfNewOrUpdate/copyIfUpdate`
Allows the scriptwriter to update a resource only if it already exists. Use the `copyIfUpdate` flag to prevent a new resource from being created. If the `copyIfUpdate` flag is used with the `keepExisting` flag then no copying will occur.

`dontIgnoreProtection/ignoreProtection`
Determines if the user should be alerted if any resource with its protected bit set will be replaced or deleted. When using the `ignoreProtection` flag the resource is deleted from or updated in the target file even if it is protected in the target file.

`srcNeedExist/srcNeedNotExist`
Determines whether the source resources must exist on the source disk. Use the `srcNeedNotExist` flag if the source resources can optionally reside on the source disk. If the

source file or source resources are not found, the atom it is ignored and the installation continues normally.

`byName/byID` Determines how the resource should be found in the source and target files. If using the `byID` flag the resource is found in the source and target file using only the ID. If you want to require the resource name to match as well then use the `nameMustMatch` flag. If using the `byName` flag the resource is found in the source and target file using only the name specified in the `Resource Name` field. Use the `byName` flag when installing Desk Accessory resources in pre-7.0 System files.

`nameNeedNotMatch/nameMustMatch`
Specifies that the resource found in the source file using its ID have the name in the Resource Name field. The resource with the same name and ID in the target file, if any, will be replaced or deleted (if the `updateExisting` flag is used). This flag is ignored if the `byName` flag, above, is used. If using the `nameNeedNotMatch` flag the `byID` flag must also be used.

Field Descriptions

`Total Target Size` The size, in bytes, of the copied target resource and all its owned resources. This field is used only by Installer Engine in figuring the disk size needed for an installation. (4-bytes)

`Target Resource File Spec.`
The resource ID of a Target File Spec script resource (‘intf’, or ‘infs’ for pre-4,0 scripts) describing the file on the target disk where the resource will be deleted, created, or updated. (2-bytes)

`Target Resource Type` The resource type Installer Engine will use to find or create the resource in the target file. (4-bytes)

`Target Resource ID` The resource ID Installer Engine will use to find or create the resource in the target file. If the ID of the resource doesn’t matter and you’re copying using the `byName` flag (e.g., most Desk Accessories), this field should be 0, in which case Installer Engine picks an ID for the resource in the target file. (2-bytes)

`Target Resource Attributes`
The attributes that will be given to the target resource when copying. (2-bytes)

`Target Resource Name` The name that will be given to the target resource. If this string is empty, the name of the first source resource will be given to the target resource. (even-padded Pascal string)

`Source Piece File Spec.` The resource ID of a Source File Spec. (‘infs’) describing the source file that holds the resource part. If you’re just deleting a resource and therefore don’t need a source file, leave the source list empty. (2-bytes)

`Source Piece Type` The resource type of the resource piece being copied. If there is more then one piece, it's recommended that the type by 'part'. (4-bytes)

`Source Piece ID` The resource ID of the resource piece being copied. This field is ignored if the resource piece is being found by name (`byName` flag is used). (2-bytes)

`Target Piece Size` The number of bytes the resource piece will occupy in the target file. If the resource will be decompressed using an Atom Extender, then this value should be the noncompressed size of the source piece. This value is used to compute where multiple pieces are to be written in the final target file, and therefore must be exact. (4-bytes)

`Source Piece Resource Name`
The name of the source resource. This field is only needed when finding the source resource by its resource name (`byName` flag is used). (even-padded Pascal string)

`Source Version Number` The version number (BCD format) of the source resource when a Version Compare function is used to calculate the target version number. (4-bytes)

`Version Compare Rsrc ID` The resource ID of a Version Compare ('invc') script resource. The resulting version number from the version function will be compared with the `Source Version Number` field to determine if the target is newer or older than the source. If this field is 0 (zero) then the target version number is assumed to be 0 (zero). (2-bytes)

`Atom Extender Rsrc ID` The resource ID of an Atom Extender ('inex') script resource. This Atom Extender will be called during copying of each resource part. (2-bytes)

When using InstaCompOne compression enter a value of 241 here.

`Resource Atom Description`

The `Resource` Atom Description field is an even-padded Pascal string describing the atom. This is used as part of the status dialog. If you do not supply a description Installer Engine will display "Building File: [file name]" or "Writing File: [file name]." If you supply a description it will be appended to the string "Building " or "Writing ".

Resource Atom List Resource ('inr#')

```
type 'inr#' {
       switch {
           case format0:
               key integer = 0;      /* Resource Atom Format version */

               integer = $$CountOf (inraRsrcs); /* Number of inraRsrcs */
               wide array inraRsrcs {
                   integer;                     /* Resource ID */
                   resourceAtomFlagsFormat1;    /* Resource Atom Flags */
```

```
                unsigned longInt;           /* Total Target Size  */
                fileSpecID;                 /* Target File Spec */
                fileSpecID;                 /* Source File Spec */
                rsrcType;                   /* Src/Tgt Rsrc Type */
                rsrcID;                     /* Src/Tgt Rsrc ID */
                integer;                    /* Tgt Rsrc Attributes */
                unsigned longint;           /* Source Version Number */
                rsrcID;                     /* Version Compare Rsrc ID */
                rsrcID;                     /* Atom Extender ID */
                unsigned longInt;           /* Target Size of Rsrc */
            };
        };
};
```

Flag Descriptions

The flags for the Resource List Atom are identical to format 1 of the Resource Atom. Since there is no source name field in the Resource List Atom you should always use the `byID` and `nameNeedNotMatch` flags.

Field Descriptions

`Resource ID` — The ID value that will be used to find this record from the reference in a package. For example, if you choose the ID value of 369, then the reference from a package should have the type ‘inra’ and the ID 369. (4-bytes)

See field descriptions of ‘infa’ resource for information on the other fields

Using the Font Atom

The Font Atom is used to describe a set of font strikes (size/style) from a specified font family that should be copied to or removed from a file. NOTE: Apple now copies fonts as entire suitcases using File Atoms instead of the Font Atom. We encourage developers to do the same.

The Font Atom provides the following major features for copying or deleting fonts:

- Ability to preserve an existing font strike.
- Ability to specify an individual strike to copy/delete or simple mode that copies the complete family.
- Support for installing into all versions of System Software with a single Font Atom.
- Decompression of font resources (‘NFNT’, ‘sfnt’ or ‘FONT’) during installation.
- Installation of a font strike that has been split among multiple resources on multiple source disks.

Note: Apple recommends installing fonts as entire suitcase files instead of using the font atom

Auto-Routing Under Pre-7.1 Systems

To overcome the effort involved with installing fonts into the various versions of System Software, Installer Engine will automatically place the font resources into the System file if installing into a System Folder that does not support the “Fonts” folder. To use this feature, write your script to always install into a file in the System Folder’s “Fonts” folder. Make the target file path start with the reserved folder name “special-font” and reference this target file spec. from the Font Atom, and Installer Engine will handle the rest.

Using Atom Extenders with Font Atoms

You'll use an Atom Extender to decompress font resources that were compressed by you to save space on your source disks. Minor constraints are required when using the Font Atom with this feature.

Scriptwriters must specify individual strikes(size and style) to copy when using an Atom Extender to decompress font resources. This is required because the target piece sizes for each strike must be entered in the optional source list. It's recommended to place compressed font resource data into resources of type 'part'.

The Font Atom is expanded into one or more parts that the Atom Extender receives with the `kBeforePart` and `kAfterPart` messages. As each part is installed, the Atom Extender attached to the original Font Atom will be called for each part. There will be a part for each source resource piece being copied. The only exception to this rule is for strikes that reference font resources in ROM. Since there is no data copied for these strikes the Atom Extender is not called.

NOTE

The 'FOND' resource is always copied using Installer Engine's default copy mechanism. This prevents the scriptwriter from decompressing the 'FOND' resource using an Atom Extender. The 'FOND' resource is the road map to other font resources ('FONT, 'NFNT', and 'sfnt'), and therefore must never be compressed. We suggest that you convert the 'FOND' resource into the type 'iFND' and use the `encodedFONDRsrc` flag when compressing any of the font resources it references. This will prevent the system software from becoming confused when finding a 'FOND' resource in your source file that may reference not existent font resources. In addition, the `compressed` flag should NOT be set in the resource attributes of the 'part' resource because this flag is presently reserved for use by Apple Computer, Inc. ◆

Font Atom Reference

This section describes the resource description of the 'inff' script resource, format 1.

Resource Description

```
#define Style                                                          \
    fill bit[9];                                    /* Reserved */     \
    Boolean noExtendedStyle, extendedStyle;     /* Extended style */   \
    Boolean noCondensedStyle, condensedStyle;   /* Condensed style */  \
    Boolean noShadowStyle,    shadowStyle;      /* Shadow style */     \
    Boolean noOutlineStyle, outlineStyle;       /* Outline style */    \
    Boolean noUnderlineStyle, underlineStyle;   /* Underline style */  \
    Boolean noItalicStyle, italicStyle;         /* Italic style */     \
    Boolean noBoldStyle, boldStyle;             /* Bold style */

#define fontFamilyAtomFlags                                            \
       boolean    dontDeleteWhenRemoving, deleteWhenRemoving;          \
       boolean    dontDeleteWhenInstalling, deleteWhenInstalling;      \
       boolean    dontCopy, copy;                                      \
       fill bit[5]                                  /* Reserved */    \
       boolean    noEncodedFONDRsrc, encodedFONDRsrc;                  \
       boolean    noTgtRequired, tgtRequired;                          \
       boolean    updateExisting, keepExisting;                        \
       boolean    copyIfNewOrUpdate, copyIfUpdate;                     \
       boolean    dontIgnoreProtection, ignoreProtection;              \
       boolean    srcNeedExist, srcNeedNotExist;                       \
       boolean    byName, byID;                                        \
```

```
        boolean    nameNeedNotMatch, nameMustMatch

#define RsrcSpec                                                    \
        fileSpecID;             /* Source Piece File Spec. */       \
        rsrcType;               /* Source Piece Rsrc Type*/         \
        rsrcID;                 /* Source Piece Rsrc ID */          \
        unsigned longInt;       /* Source Piece Rsrc Size */        \
        evenPaddedString        /* Source Piece Rsrc Name */

#define SrcPartsList                                                \
        integer = $$CountOf (Pieces);/* Number of source pieces */  \
        wide array Pieces{                                          \
            RsrcSpec;               /* Description of this piece*/  \
        }

#define Strike                                                      \
    integer;                /* Font Size */                         \
    Style;                  /* Font Style */                        \
    RsrcType;               /* Target Font Resource Type */         \
    integer;                /* Target Attributes */                 \
    SrcPartsList;           /* Optional source pieces */

type 'inff' {
    switch {
        case format2:
            key integer = 2;          /*format version 2 */
            fontFamilyAtomFlags;      /* Font Atom Flags */
            fileSpecID;               /* Target File Spec */
            fileSpecID;               /* FOND Source File Spec */
            integer;                  /* Target FOND Attributes */
            unsigned longInt;         /* Family Size */
            rsrcID;                   /* Target Family Number */
            switch {
            case entireFamily:
                key integer = 1;
            case explicitFamilyMembers:
                key integer = 2;
                unsigned integer = $$CountOf(StrikeEntries);
                wide array StrikeEntries {
                    Strike;           /* Source for each of the strikes */
                };
            };
            rsrcID;                   /* Atom Extender ID */
            evenPaddedString;         /* Atom Description */
            evenPaddedString;         /* Family Name */
    };
};
```

Flag Descriptions

`dontDeleteWhenRemoving/deleteWhenRemoving`

Determines if specific font strikes or the entire family is deleted during a removal.

Using with `explicitFamilyMembers` copy option — When using the `deleteWhenRemoving` flag, if the target font strike exists and the user clicks Remove the strike is deleted. If a strike does not exist on the target disk, this flag is ignored. If all font strikes for a given family are removed, the family's 'FOND' resource is also deleted.

Using with `entireFamily` copy option — When using the `deleteWhenRemoving` flag, the family's 'FOND' resource and *all* font strikes for the family are deleted.

Note that the font atom must be part of a package that uses the `removable` flag.

`dontDeleteWhenInstalling/deleteWhenInstalling`

Determines if specific font strikes or the entire family is deleted during an installation when using the `dontCopy` flag.

Using with `explicitFamilyMembers` copy option — When using the `deleteWhenRemoving` flag, if the target font strike exists and the user clicks Remove the strike is deleted. If a strike does not exist on the target disk, this flag is ignored. If all font strikes for a given family are removed, the family's 'FOND' resource is also deleted.

Using with `entireFamily` copy option — When using the `deleteWhenRemoving` flag, the family's 'FOND' resource and *all* font strikes for the family are deleted.

The `deleteWhenInstalling` flag is primarily used for deleting previously installed font resources that are no longer needed.

`dontCopy/copy` — Determines if the font resources are copied or not during an installation. Note that some flags (`leaveAloneIfNewer`, `keepExisting`, `copyIfUpdate`) can prevent copying from happening under the circumstances specified.

`noEncodedFONDRsrc/encodedFONDRsrc`

Allows 'FOND' resources to be encoded into the resource type 'iFND' when compessing the font resources. This is recommended to help prevent the system software from becoming confused when finding a 'FOND' resource in your source file that may reference not existent font resources.

`noTgtRequired/tgtRequired`

The target file must already exist. If the target file does not exist, the user is warned that a target file is needed. When using the `noTgtRequired` flag and the target file does not exist, Installer Engine will create one.

`updateExisting/keepExisting`

Allows the scriptwriter to preserve existing font resources when using the `copy` and/or `deleteOnInstall` flag. Use the `keepExisting` flag to prevent Installer Engine from disturbing an existing target font strike or family.

Using with `explicitFamilyMembers` copy option — When using the `keepExisting` flag and a specific strike in the 'FOND' resource already exists in the target file, it will not be replaced.

Using with `entireFamily` copy option — When using the `keepExisting` flag and the 'FOND' resource already exists, no strikes will be copied or replaced.

No copying will occur if the `updateExisting` flag is used with the `copyIfUpdate` flag.

`copyIfNewOrUpdate/copyIfUpdate`

Allows the scriptwriter to update a font resource only if it already exists. Use the `copyIfUpdate` flag to prevent a new resource from being created.

Using with `explicitFamilyMembers` copy option — When using the `copyIfUpdate` flag, a specific strike is copied to the target file only if the strike already exists in the target file.

Using with `entireFamily` copy option — The entire source font family will be copied if any strikes exist in the target file.

If the `copyIfUpdate` flag is used with the `keepExisting` flag, then no copying will occur.

`dontIgnoreProtection/ignoreProtection`

Determines if the user should be alerted if any font resource with its protected bit set will be replaced or deleted.

Using with `explicitFamilyMembers` copy option — When using the `dontIgnoreProtection` flag, if the 'FOND' resource or any font resource to be replaced has the protection flag set, the user will be alerted, and the installation will be canceled. Use the `dontIgnoreProtection` flag if preserving protected font resources is important. When using the `ignoreProtection` flag the font resources are deleted from or updated in the target file even if it any are protected in the target file.

Using with `entireFamily` copy option — When using the `dontIgnoreProtection` flag the user is alerted only if the 'FOND' resource is protected, but if any other font resources are protected, they will be replaced without alerting the user. When using the `ignoreProtection` flag the font resources are deleted from or updated in the target file even if it any are protected in the target file.

`srcNeedExist/srcNeedNotExist`

Determines whether the source font resources must exist on the source disk. Use the `srcNeedNotExist` flag if the source resources can optionally reside on the source disk. If the source file or source resources are not found, the atom it is ignored and the installation continues normally.

`byName/byID`

Determines how the font strike resources should be found in the source files. If using the `byID` flag the font strike's resources are found in the source file using only the ID. If you want to require the resource name to match as well then

use the `nameMustMatch` flag. Note that the `byID` flag can only be used when installing using the `explicitFamilyMembers` copy option and split source resources are specified. If using the `byName` flag the font strike resources are found in the source file using only the name specified in the `Source Piece Rsrc Name` field.

`nameNeedNotMatch/nameMustMatch`
Specifies if the font strike resources found in the source file using their IDs must have the name in the `Source Piece Rsrc Name` field. Note that the `nameMustMatch` flag can only be used when installing using the `explicitFamilyMembers` copy option and split source resources are specified. This flag is ignored if the `byName` flag is used.

Field Descriptions

`Target File Spec ID` The resource ID of a Target File Spec script resource (‘intf’, or ‘infs’ for pre-4,0 scripts) describing the file on the target disk where the resource will be deleted, created, or updated. (2-bytes)

`Source File Spec ID` The resource ID of a Source File Spec script resource describing the file on the source disk that contains the 'FOND' resource to be copied into the target file.

Using with `explicitFamilyMembers` copy option — If no optional split source information is provided, the strikes specified must also be contained in the same file as the source ‘FOND’ resource. Optional split source information always overrides that default source for those strikes that use it.

Using with `entireFamily` copy option — The source strike resources referenced in the source 'FOND' resource must be contained in the same source file as the source 'FOND' resource.

If you’re just deleting strikes and therefore don’t need a source file, set this field to 0. (2-bytes)

`Target FOND Attributes` The resource attributes that will be given to the target 'FOND' resource during installation. See Inside Macintosh Vol. I, page 111 for more information about resource attributes. (2-bytes)

`Font Family Size` The size in bytes of all font resources including the 'FOND' resource to be installed or deleted. This field is used by Installer Engine in figuring the sizes for an installation. (4-bytes)

`Target FOND ID` The resource ID that will be given to the target 'FOND' resource. Normally, this ID should also be the same ID as the resource ID of the source 'FOND' resource. Installer Engine always sets the family number field contained in the ‘FOND’ resource to the resource ID of the ‘FOND’ when it is copied or updated. (2-bytes)

How-To-Copy Value Key	Specifies how the font resources will be copied, or deleted. Currently, the two copy options are available: `entireFamily` Value (1), and `explicitFamilyMembers` Value (2). (2-bytes) **Using the `entireFamily` copy option** — Tells Installer Engine that all strikes specified in the source 'FOND' resource will be copied. The 'FONT', 'NFNT', or 'sfnt' resources must exist in the same source file as the 'FOND' resource. The Copy Entire Family Value is provided to copy small, simple font families with minimal scripting effort. Since the target resource attributes cannot be specified, the target resource inherits the source's. If the Font Family Atom Flags are set for `deleteWhenRemoving` or `deleteWhenInstalling & dontCopy` the target 'FOND' resource and ALL of its referenced strikes will be deleted from the target file upon removal or installation. (2-bytes) **Using the `explicitFamilyMembers` copy option** — Tells Installer Engine that one or more strikes are specified in the following member list. (2-bytes)
Point Size	The family member's point size. (2-bytes)
Style	Contains flags describing the style of the font. The style bits are defined below. (2-bytes) Bit 0: Set for Bold Bit 1: Set for Italic Bit 2: Set for Underline Bit 3: Set for Outline Bit 4: Set for Shadow Bit 5: Set for Condensed Bit 6: Set for Extend All other bits are reserved.
Resource Type	The resource type of the target or source font resource. This will be the type of the font resource created by combining the split resources. If no split resource information is given, this field should be the type of the source font resource (e.g. `'FONT', 'NFNT', or 'sfnt'`). (4-bytes)
Resource Attributes	The resource attributes that will be given to the target font resource. The Split Resource list specifies one or more resources that make up the font resource when combined in the order listed. (2-bytes)
Split Resource File Spec ID	The Split Resource File Spec ID field is a 2-byte field that specifies the file on the source disk that contains the split resource to be combined. (2-bytes)
Split Resource Type	The resource type of the source split resource. Usually, this should be type 'part'. (4-bytes)
Split Resource ID	The resource ID of the source split resource. (2-bytes)
Split Resource Size	The exact size in bytes of the source split resource. (4-bytes)

`Split Resource Name`	The name of the source split resource. If the resource is being found using the `byName` flag, this field must be filled. To help ensure that the correct resource is being installed, the name of the resource in the source file must match this field if the resource is being found by name. (even-padded Pascal string)
`Atom Extender ID`	The resource ID of an Atom Extender ‘inex’ script resource. This Atom Extender will be called during copying of each font resource part. (2-bytes)
`Font Family Atom Description`	A Pascal string describing the atom. This field is used as part of the status dialog. If this field is not empty, the status dialog displays “Reading <Description String>”, or “Writing <Description String>”. If this field is empty, the status dialog displays “Reading font: <Font Family Name>”, or “Writing font: <Font Family Name>”. (even-padded Pascal string)
`Font Family Name`	The family name. The family name will be used to find the source and target 'FOND' resource. (even-padded Pascal string)

Using the ResMerge Atom

The ResMerge Atom can be used to copy all resources from a source file to a target file. Each resource is copied separately, replacing an existing resource of the same type and ID. Because the ResMerge Atom offers no options, you may find that you will need to use individual Resource Atoms to handle special situations.

The ResMerge Atom is unique from using Resource Atoms because anyone can add to or delete resources from the source file without changing the atom. This is acceptable as long as Installer Engine can successfully preflight the required disk space. A field in the atom holds the total size the resources expect to occupy in the target file. This flexibility comes at a cost to ease of use because Installer Engine cannot determine during preflighting how many resources will be replaced, thus not requiring additional disk space. We will therefore usually overestimate the required disk space, forcing the user to make more space available than will probably be necessary.

The ResMerge Atom offers limited options, so determining when to use the ResMerge Atom versus separate Resource Atoms or a File Atom is important. The main purpose of the ResMerge Atom is to provide a simple high-level resource copy mechanism without forcing the scriptwriter to create numerous Resource Atoms.

Installer Engine executes a ResMerge Atom by automatically converting the atom into individual Resource Atoms when Installer Engine is ready to begin reading from the source disk. The Resource Atom is created with the following flags:

- dontDeleteWhenRemoving — The ResMerge Atom does nothing during a removal.
- dontDeleteWhenInstalling.
- copy.
- noTgtRequired — The target file will be created if it does not already exist.

- updateExisting — Existing resources with the same type and ID will be replaced.
- copyIfNewOrUpdate.
- srcNeedNotExist — Doesn't really matter because if it didn't exist we would not be creating this Resource Atom to copy it.
- byID.
- nameNeedNotMatch — The name is ignored, but the new name will replace the old name if different.

NOTE

System Software scriptwriters must use care when using the ResMerge Atom because on pre-7.0 systems we determine if the Folder Manager will be available based on the presence of a Resource Atom copying a 'fld#' resource. This means a separate Resource Atom must always be created to install the 'fld#' resource. ◆

ResMerge Atom Reference

The format of the ResMerge Atom is simple. Its template is shown below.

```
#define resMergeAtomFlags                                          \
        fill bit[16];

type 'inrm' {
        switch {
            case format0:
                key integer = 0;          /* ResMerge Atom Format version */
                mergeAtomFlags;           /* ResMerge Atom Flags */
                unsigned longInt;         /* Total Resources Size */
                fileSpecID;                  /* Target File Spec. Rsrc ID */
                fileSpecID;                  /* Source File Spec. Rsrc ID */
                evenPaddedString;         /* Status Description */
        };
};
```

Field descriptions

`ResMerge Atom Flags` Currently reserved for use by Apple Computer, Inc. (2-bytes)

`Total Resources Size` The size that all resources will occupy in the target file. For most purposes, the size of the source file's resource fork is an adequate estimate. This field is used by Installer Engine to figure the amount of disk space required. (4-bytes)

`Target File Spec. Rsrc ID` The resource ID of a Target File Spec ('intf') script resource describing the desired location of the target file. (2-bytes)

`Source File Spec. Rsrc ID` The resource ID of a Source File Spec ('infs') script resource describing the location of the source file. (2-bytes)

`Status Description` An optional string that is displayed in the Status dialog during execution of the ResMerge Atom. (even-padded Pascal string)

Using the Folder Atom (‘infm’)

The Folder Atom can be used to copy all files from the root-level of a source folder to the root-level of a target folder. Each file is copied separately, replacing an existing file of the same name. Folders within the source folder are ignored, and must be copied using separate Folder Atoms. Because the Folder Atom offers no options, you may find that you will need to use individual Resource Atoms to copy some or all the resources.

The Folder Atom is unique from using File Atoms because anyone can add to or delete files from the source folder without changing the atom. This is OK as long as Installer Engine can successfully preflight the required disk space. A field in the atom holds the total size the source files. This flexibility comes at a cost to ease of use because Installer Engine cannot determine during preflighting how many files will be replaced, thus not requiring additional disk space. We will therefore usually overestimate the required disk space, forcing the user to make more space available than will probably be necessary.

NOTE

The Folder Atom only copies the files at the root level of the source folder. Nested folders must be copied separately with additional Folder Atoms. ◆

The Folder Atom offers no options, so determining when to use the Folder Atom versus separate File Atoms is important. The main purpose of the Folder Atom is to provide a simple high-level file copy mechanism without forcing the scriptwriter to create numerous File Atoms. The Folder Atom does not support decompression or version comparison.

Installer Engine executes a Folder Atom by automatically converting the atom into individual File Atoms when Installer Engine is ready to begin reading from the source folder. The File Atom is created with the following flags:

- dontDeleteWhenRemoving — The Folder Atom does nothing during a removal.
- dontDeleteWhenInstalling.
- copy.
- dontIgnoreLockedFile — The user is notified about locked target files, and the installation is stopped.
- dontSetFileLocked or setFileLocked — If the source file is locked it will be locked after it is copied.
- useSrcCrDateToCompare — Since no version compare procedure can be specified.
- srcNeedNotExist — Doesn’t really matter because if it didn’t exist we would not be creating this File Atom to copy it.
- rsrcForkInRsrcFork — Since no decompression is allowed.
- updateExisting — Existing files with the same name will be replaced.
- copyIfNewOrUpdate.
- rsrcFork — Copy resource fork if it exists.
- dataFork — Copy data fork if it exists.

Specifying the Source and Target Folder

The standard ‘infs’ and ‘intf’ file spec resources are used to specify the source and target folder with several differences:

- The path name is similar to normal file specs, but the path should end with the name of the source or target folder. A colon at the end of the path is optional.
- All File Spec flags are ignored when used with Folder Atoms.
- The File Type, File Creator and Creation Date fields are ignored when used with Folder Atoms.

NOTE

The reserved folder path name “special-xxxx” cannot be used with Folder Atoms. If you do, the user will be told the script document is damaged. ◆

Installing Folders with Custom Icons

The Folder Atom will automatically set the target folder to use the source folder’s custom icon if one exits. Although the scriptwriter need not be aware of how the custom icon is stored, Installer Engine uses the File Atom’s feature of setting the target folder’s icon whenever the special invisible “Icon\n” file is written to disk.

Creating Empty Folders with a Folder Atom

Installer Engine will not create a target folder if there are no files are present in the source file. If you wish to create an empty folder, the easiest work around is to create a custom icon for the source folder, thereby causing the invisible “Icon\n” file to be copied, and the target folder to be created.

Folder Atom Reference

The format of the Folder Atom is simple. Its template is shown below.

```
#define folderAtomFlags                                                 \
       fill bit[16];

type 'infm' {
       switch {
           case format0:
               key integer = 0;         /* Folder Atom Format version */
               folderAtomFlags;         /* Folder Atom Flags */
               unsigned longInt;        /* Total Folder Size */
               rsrcID;                  /* Target File Spec. Rsrc ID */
               rsrcID;                  /* Source File Spec. Rsrc ID */
               evenPaddedString;        /* Status Description */
       };
};
```

Field Descriptions

`Folder Atom Flags` Currently reserved for use by Apple Computer, Inc. (2-bytes)

`Total Folder Size`	The size of all files in the source folder. This field is used by Installer Engine to figure the amount of disk space required. (4-bytes)
`Target File Spec. Rsrc ID`	The resource ID of a Target File Spec (‘intf’) script resource describing the desired location of the target folder. (2-bytes)
`Source File Spec. Rsrc ID`	The resource ID of a Source File Spec (‘infs’) script resource describing the location of the source folder. (2-bytes)
`Status Description`	An optional string that is displayed in the Status dialog during execution of the Folder Atom. (even-padded Pascal string)

Using Action Atoms (‘inaa’)

Action Atoms are used to run a code resource at the beginning and/or end of an installation or removal.

To use an Action Atom, an `'inaa'` resource must first be added to the script. This atom specifies the resource which contains an executable piece of code. The user-defined Action Atom code can be executed at two different times for installations. One is after the user clicks on the Install button but before installation begins. The other is after the installation is finished, but before the user is asked to quit or continue. The code can also be executed at two different times for removals. One is after the user clicks on the Remove button but before removal begins. The other is after the removal is finished, but before the user is asked to quit or continue. For installation and removal the order of execution is defined by the ID of the `'inaa'` resources.

NOTE: You can no longer display dialogs or interact directly with the user when your code resource is running under Installer Engine. Please see the chapter “Run-Time Issues” for information about this limitation.

Action Atom Reference

This section describes the Action Atom resource format, data structures and function interface.

Resource Description

The format of the Action Atom is shown below.

```
#define actionAtomFlagsFormat2                                          \
       fill bit[12];                                                    \
       boolean    continueBusyCursors,  suspendBusyCursors              \
       boolean    actAfter, actBefore;                                 \
       boolean    dontActOnRemove, actOnRemove;                        \
       boolean    dontActOnInstall, actOnInstall;

type 'inaa' {
       switch {
           case format2:
               key integer = 2;         /* Action Atom Format version. */
               actionAtomFlagsFormat2;  /* Action Atom Flags */
               literal longint;         /* Code Resource Type */
               integer;                 /* Code Resource ID */
```

```
            longint;                /* RefCon */
            longint;                /* Requested Memory (in bytes) */
            evenPaddedString;       /* Status Description */
        };
};
```

Flag Descriptions

`continueBusyCursors/suspendBusyCursors`
NOTE: Installer Engine ignores this flag because Installer Engine script can not display dialogs directly.

`actAfter/ActBefore` Determines whether this Action Atom code resource is called before or after the installation/removal.

`dontActOnRemove/actOnRemove`
Determines if this Action Atom code resource should be called on a removal.

`dontActOnInstall/actOnInstall`
Determines if this Action Atom code resource should be called on an installation.

Field descriptions

`Code Resource Type` The resource type of the code resource, usually 'infn'. (4-bytes)

`Code Resource ID` The resource ID of the code resource. (2-bytes)

`RefCon` A value to be passed directly to the code resource. For example, the System 7.X Installer script has all of its Action Atoms linked together into one code resource and uses this refCon value as a selector that tells which action atom to run. By linking all action atoms together into one code resource and eliminating duplicate code, we cut the size of our action atoms in half. (4-bytes)

`Requested Memory` The requested number of bytes that should be available in the Action Atom's sub-heap when called. Enter 0 (zero) to not create a sub-heap and run inside Installer Engine's heap. Please see warnings about using sub-heaps in the Atom Extender section if you specify a value other than 0. (4-bytes)

`Status Description` An optional string that is displayed in the Status dialog during execution of the Action Atom. (even-padded Pascal string)

Data Structures

Function Interface

The entry point of the Action Atom code resource must have the interface:

```
ActionAtomResult  ActionAtomFormat2( ActionAtom2PBPtr );
```

The Action Atom should return a result telling Installer Engine what to do after executing the code resource. You can use these constants to specify the result code:

```
enum {     kActionAtomResultFatalError = -1,
           kActionAtomResultContinue = 0,
           kActionAtomResultCancel = 1 };

typedef long ActionAtomResult;
```

Return results produce the following effect:

- Return `kActionAtomResultContinue` if you wish the installation to continue.
- Return `kActionAtomResultCancel` to cancel the installation, just as if the user had pressed the Cancel button. When an action atom returns the cancel message, all action atom waiting in the queue are not run. Cancel messages are then sent to all 'after' action atoms to allow them to clean up their environment.
- Return `kActionAtomResultFatalError` to signal that there is something seriously wrong, and the installation should not continue. When an action atom returns the error message, all action atoms waiting in the queue are not run. Cancel messages are then sent to all 'after' action atoms to allow them to clean up their environment.

▲ **WARNING**

If you choose to create a sub-heap for each invocation of format 2 of the Action Atom please read the section "Running with a Sub-Heap" within the Atom Extender portion of this document. You can enter 0 (zero) in the requested memory field to not create a sub-heap and run inside Installer Engine's heap. ▲

Parameter Block

The Action Atom parameter block contains information about Installer Engine's environment. The parameter block has the structure:

```
typedef struct {
    InstallationStage fMessageID;
    Handle            fStaticDataHdl;
    ProcPtr           fCallBackProcPtr;
    short             fTargetVRefNum;
    long              fTargetFolderDirID;
    short             fSystemVRefNum;
    long              fSystemBlessedDirID;
    long              fRefCon;
    Boolean           fDoingInstall;
    Boolean           fDidLiveUpdate;
    long              fInstallerTempDirID;
}ActionAtom2PBRec, *ActionAtom2PBPtr;
```

Field Descriptions

`fMessageID` One of three messages the Atom Extender will receive when being called. Use these constants to understand the message ID:

```
enum {before,
   after,
   cleanUpCancel };
```

```
typedef unsigned char InstallationStage;
```

If your Action Atom is running before the installation takes place, the `fMessageID` field is `before`.

If your Action Atom is running after the installation completes successfully, the `fMessageID` field will be `after`.

If the user clicks the Cancel/Stop button, or Installer Engine cancels the installation because of an error, `cleanUpCancel` messages are sent to all 'after' action atoms to allow them to clean up their environment. Each 'after' action atom is guaranteed to receive only one cancel message if the installation is canceled or stopped. This may happen before the 'before' actions atoms have run, so it's important to be smart within your cleanup routines.

`fStaticDataHdl` — A handle created by the Action Atom using INewHandle to save information between before and after calls to this Action Atom. The field is always NULL when receiving a `before` message. You can assign a value to this field during the first call to the Action Atom, and receive the same value in this field during the next call.

`fCallBackProcPtr` — A pointer to Installer Engine's dispatch routine. You'll need to pass this field as a parameter to glue routines that provide access to Installer functions.

`fTargetVRefNum` — The target disk's vRefNum. When allowing the user to select a target application folder, this is the volume on which the folder resides.

`fTargetFolderDirID` — The target application folder's directory ID. Target File Specs that use the reserved folder path `folder-user` will be placed in this folder. This value is -1 if Installer Engine is using disk mode, the folder does not exist, or a File Spec referencing the reserved folder path `folder-user` has not been referenced by any atom included in the installation.

`fSystemVRefNum` — The system disk's vRefNum. Target File Specs that use the reserved folder path `special-xxx` will be placed in the System Folder on this volume.

`fSystemFolderDirID` — The directory ID of the System Folder on the disk with the refNum `fSystemVRefNum`. This directory is not necessarily the currently active System Folder.

`fRefCon` — A 4-byte value defined by the scriptwriter in the 'inex' script resource.

`fDoingInstall` — TRUE if the user is performing an installation; otherwise, FALSE if the user is removing.

`fDidLiveUpdate` — TRUE if Installer Engine is modifying files inside the active System Folder.

`fInstallerTempDirID` — The directory ID of the temporary folder on the volume specified in the `fSystemVRefNum` field. The temporary

folder holds the files that we are modifying to allow rollback in case the installation is canceled. You'll rarely ever need to look inside this folder.

Using Audit Atoms ('inat')

During the development of System Software scripts, there were many times we wished that there was a history of what had previously been installed. For example, the user did a minimal SE System Software installation, and later wanted to add minimal Macintosh II System Software. During the second installation, Installer Engine should be smart enough to update the disk with both SE and II software. Installer Audit Atoms were added to facilitate this type of updating. Their format is shown below.

When Installer Engine finds an Audit Atom in a script, after an installation is complete, it adds a new resource (type `'audt'`) to the target file specified in the Audit Atom. This resource contains an array of audit selector/value pairs. If an `'audt'` resource already exists in the target file and there is no entry for the given selector, an entry is added. If an entry for the selector already exists, the higher of the two values is used. Using Installer Rules `checkAuditRecord` and `checkAnyAuditRecord`, later installations can make decisions based on this history.

Audit Atom Reference

Resource Description

Format 0 of the 'inat' resource:

```
type 'inat' {
    switch {
        case format0:
            key integer = 0;
                FileSpecID;          /* Target File Spec Rsrc ID*/
                OSType;           /* Audit Selector */
                literal longint;  /* Audit Value */
            };
};
```

Field descriptions

`Target File Spec. Rsrc ID`
: The resource ID of a Target File Spec. of which you want to add or update the audit resource. Apple uses the System File to keep a history of System Software that has been installed. A File Spec (`'intf'` resource) must exist in the script with this ID. (2-bytes)

`Audit Selector`
: A meaningful type. (4-bytes)

`Audit Value`
: A meaningful value. (4-bytes)

Using Boot Block Atoms (‘inbb’)

Boot Block Atoms are used to write or change the parameters in a target volume’s boot blocks. Boot Block Atoms are different from file and resource atoms – they can indicate something to be copied, or indicate an individual parameter that needs to be changed.

To cause boot blocks to be written to a target volume, include a Boot Block Atom with a key of type `'bbUpdate'`. The argument to this type of Boot Block Atom is an integer which indicates the ID of a File Spec in the script. The file indicated by this File Spec should contain a `'boot'` resource. A copy of the resource is written to the first two blocks of the target volume.

Boot Block Atom Reference

Resource Description

Format 0 of the ‘inbb’ resource:

```
#define BootBlockAtomFlags                              \
    fill bit[14];                                       \
    boolean dontChangeOnInstall, changeOnInstall;       \
    boolean dontChangeOnRemove, changeOnRemove;

#define BootBlockUpdateFlags                            \
    fill bit[7];                                        \
    boolean replaceBBSysName, saveBBSysName;            \
    boolean replaceBBShellName, saveBBShellName         \
    boolean replaceBBDbg1Name, saveBBDbg1Name;          \
    boolean replaceBBDbg2Name, saveBBDbg2Name;          \
    boolean replaceBBScreenName, saveBBScreenName;      \
    boolean replaceBBHelloName, saveBBHelloName;        \
    boolean replaceBBScrapName, saveBBScrapName;        \
    boolean replaceBBCntFCBs, maxBBCntFCBs;             \
    boolean replacebbCntEvts, maxBBCntEvts              \

type 'inbb' {
    switch {
        case format0:
            key integer = 0;
            BootBlockAtomFlags;
                switch {
                    case bbUpdate:
                        key integer = -1;
                        RsrcID;
                        BootBlockUpdateFlags;

                    case bbID:
                        key integer = 1;
                        decimal integer;

                    case bbEntry:
                        key integer = 2;
                        decimal longint;

                    case bbVersion:
                        key integer = 3;
                        decimal integer;

                    case bbPageFlags:
                        key integer = 4;
```

```
                    decimal integer;

                case bbSysName:
                    key integer = 5;
                    EvenPaddedString;

                case bbShellName:
                    key integer = 6;
                    EvenPaddedString;

                case bbDbg1Name:
                    key integer = 7;
                    EvenPaddedString;

                case bbDbg2Name:
                    key integer = 8;
                    EvenPaddedString;

                case bbScreenName:
                    key integer = 9;
                    EvenPaddedString;

                case bbHelloName:
                    key integer = 10;
                    EvenPaddedString;

                case bbScrapName:
                    key integer = 11;
                    EvenPaddedString;

                case bbCntFCBs:
                    key integer = 12;
                    decimal integer;

                case bbCntEvts:
                    key integer = 13;
                    decimal integer;

                case bb128KSHeap:
                    key integer = 14;
                    decimal longint;

                case bb256KSHeap:
                    key integer = 15;
                    decimal longint;

                case bb512KSHeap:
                    key integer = 16;
                    decimal longint;

                case bbSysHeapSize:
                    key integer = 16;
                    decimal longint;

                case bbSysHeapExtra:
                    key integer = 18;
                    decimal longint;

                case bbSysHeapFract:
                    key integer = 19;
                    decimal longint;
            };
            EvenPaddedString;        /* Boot Block Atom Description */
        };
};
```

Flags descriptions

`dontChangeOnInstall/changeOnInstall`
: Determine if the boot blocks should be updated during an installation. (2-bytes)

`dontChangeOnRemove/changeOnRemove`
: Not currently supported. (2-bytes)

Field descriptions

`Boot Block Value Key`
: Specifies which boot block parameter is being given a value in the Boot Block Value field. The key can correspond to any of the parameters which are changeable in the boot blocks. (2-bytes)

  The possible keys for this field are as follows:

`bbUpdate`
: Copy over boot blocks from a `'boot'` resource found in the file whose File Spec ID is given in the 2-byte value field. A second 2-byte value field is used for this type of Boot Block Value. This second value field is used to specify which boot block fields are to be updated from the `'boot'` resource, and which ones are to be preserved on the target disk. If bits 0 through 6 are set, the appropriate value on the target is kept if it appears to be a legal value or string. If bits 7 or 8 are set, the maximum parameter from the resource or what already exists on the target is preserved. The format of this field is as follows:

`bbID`
: The Boot Block Value field updates the boot blocks ID (2-bytes).

`bbEntry`
: The value updates the boot block entry point (4-bytes).

`bbVersion`
: The value updates the boot block version (2-bytes).

`bbPageFlags`
: The value updates the page 2 usage flags (2-bytes).

`bbSysName`
: The value updates the name of the system resource file (string).

`bbShellName`
: The value updates the name of the system shell (string).

`bbDbg1Name`
: The value updates the first loaded debugger’s name (string).

`bbDbg2Name`
: The value updates the second loaded debugger’s name (string).

`bbScreenName`
: The value updates the file name of the startup screen (string).

`bbHelloName`
: The value updates the file name of the startup program (string).

`bbScrapName`
: The value updates the file name of the system scrap file (string).

`bbCntFCBs`
: The value updates the number of FCBs to open (2-bytes).

`bbCntEvts`	The value updates the size of the event queue (2-bytes).
`bb128KSHeap`	This boot block field is no longer used.
`bb256KSHeap`	This boot block field is no longer used.
`bb512KSHeap`	The value updates the size of the system heap on a 512K Mac (4-bytes).
`bbSysHeapSize`	The value updates the absolute size of the system heap (4-bytes).
`bbSysHeapExtra`	This boot block field is no longer used.
`bbSysHeapFract`	The value updates the minimal additional system heap space required (4-bytes). Note: Under System 7.0 and greater, the system heap space and the number of FCBs to open are dynamically determined and not controlled by the boot block value.
`Boot Block Value`	The value for the boot block parameter that was specified in the boot block value key. It has a size as given above (in parentheses). (size depends on the type of update)

Using Atom Extenders (‘inex’)

Scriptwriters can use Atom Extenders to enhance or replace the default copy mechanism of Installer Engine. The most obvious purpose of the Atom Extender is to provide transparent decompression of files during installation. Scriptwriters presently using Action Atoms to perform decompression, either written by themselves or a third-party developer, will want to use Atom Extenders for this purpose in the future.

If you want to write an Atom Extender then read the section “Writing Atom Extenders” to learn how. If you only want to take advantage of an Atom Extender someone else has written then consult the sections describing the File Atom, Resource Atom and Font Atom.

An Atom Extender is a newly-defined script resource of type ‘inex’ that can be referenced from new versions of the File Atom, Resource Atom, and Font Atoms. At the heart of the Atom Extender is a code resource (provided by the scriptwriter) that contains the necessary 68K code to be executed at the desired point in the installation.

This section describes the necessary steps to create an Atom Extender.

Creating an ‘inex’ Script Resource

The new ‘inex’ script resource contains the necessary information to properly call the specified code resource.

Code Listing 3-1 Sample ‘inex’ script resource

```
resource 'inex' (129) {
    format0 {
        dontSendInitMessage,      /* Don’t send kInitialize message */
        sendBeforeMessage,        /* Send kBeforePart message */
        dontSendAfterMessage,     /* Don’t send kAfterPart message */
        dontSendSuccessMessage,   /* Don’t send kSuccess message */
        dontSendCancelMessage,    /* Don’t send kCancel message */
        continueBusyCursors,      /* Show busy cursor during call */
        'infn',                   /* Resource type of code resource */
```

```
        128,                    /* Resource ID of code resource */
        100,                    /* RefCon (long integer) */
        30720,                  /* 30K of requested free mem in heap */
        "This is a test."       /* Status description during call */
    }
};
```

The sample ‘inex’ resource in Code Listing 3-1 asks Installer Engine to call the code resource with the type ‘infn’ and ID = 128 before Installer Engine begins copying the atom’s data from which the Atom Extender is referenced. When the code resource is called it will have a maximum of 30720 bytes of free memory in its own heap, from which it can allocate memory using Memory Manager routines, such as `NewPtr` and `NewHandle`. You must check the actual size of the heap at the beginning of your code resource to make sure you were allocated the full amount, since Installer Engine calls you whether or not your requested was fully granted. This error will usually be an idication that Installer Engine’s memory partition is set too low. The actual heap size is usually 5K larger than the requested size, but you should never depend on this being true.

NOTE

Atom Extenders are only called during installations, never during removals. ◆

Writing a Simple Atom Extender

A very simple task for an Atom Extender is to simulate the default copy task of Installer Engine using the supplied Installer routines. In Code Listing 3-2 the function reads and writes the atom’s data using Installer Engine routines `ReadSourceData` and `WriteTargetData`.

Code Listing 3-2 Simple Atom Extender performing a direct copy of an atom’s data.

```
ExtenderResultCode main( ExtenderPBPtr      extenderPBPtr )
{
    OSErr                   theErr;
    long                    dataLen;
    Ptr                  theBufferPtr;
    ExtenderResultCode   resultCode;

    // -- Initalize some important variables
    dataLen = 30000;
    theErr = noErr;

    // -- Depending on the message, perform the proper task
    switch( extenderPBPtr->fFileCopyPBRec.fEnvironmentHeader.fMessageID ) {

        case kBeforePart:

            // -- Create a buffer
            theBufferPtr = NewPtr(dataLen);

            // -- Check that we got our buffer successfully
            if( theBufferPtr != NULL && MemErr() == noErr ) {

                // -- Read as much as we can up to the size of the buffer.
                theErr = ReadSourceData( &dataLen, theBufferPtr);

                // -- Loop while there is more data to read.
                while( dataLen > 0 ) {


                    // -- This is where we can massage the data before writing it out.
                    //     For example, if the source files were compressed, this is
                    //     where we might call our decompression routine.
```

```
            // -- Write the data out.
            theErr = WriteTargetData( dataLen, theBufferPtr);

            // -- Read as much as we can up to the size of the buffer.
            if( theErr == noErr ) {
                theErr = ReadSourceData( &dataLen, theBufferPtr);

                // -- If we got eofErr, then we know we're done,
                //      and dataLen will be 0 (zero).
                if( theErr == eofErr )
                    theErr = noErr;
            }

        } // while not done copying

        if( theErr == noErr )
            resultCode = kCopiedData;
        else
            resultCode = kFatalError;

        // -- Dispose the buffer
        DisposPtr(theBufferPtr);
    }
    else
        resultCode = kFatalError;

    break;

// -- For this example, we ignore the other messages.
case kInitialize:
case kAfterPart:
case kSuccess:
case kCancel:
    resultCode = kContinueAsNormal;
    break;
}

return resultCode;
}
```

Once the code in Code Listing 3-2 has been compiled into a code resource, it can be called from any atom and will copy the atom's data as if Installer Engine had performed the copy. This also works for atoms whose source pieces have been split across multiple source disks, because Installer Engine keeps track of where each source piece should be written in the target file. Installer Engine data routines are similar to high-level File Manager routines:

- `ReadSourceData` — Similar to `FSRead`.
- `WriteTargetData` — Similar to `FSWrite`.
- `SetTargetDataPos` and `SetSourceDataPos` — Similar to `SetFPos`.
- `GetTargetDataPos` and `GetSourceDataPos` — Similar to `GetFPos`.
- `GetTargetDataEOF` and `GetSourceDataEOF` — Similar to `GetEOF`.

One of five messages can be sent to the Atom Extender. Flags in the 'inex' resource specify which messages the Atom Extender wishes to receive. The Atom Extender writer can examine the `fMessageID` field in the parameter block to determine which message has been sent. The following messages have been defined:

- `kInitialize` — Sent after the user clicks the Install button and preflighting has successfully completed.

- kBeforePart — Sent after Installer Engine has opened the atom's source file and is about to copy the atom's data.
- kAfterPart — Sent after the atom's data has been successfully copied.
- kSuccess — Sent after the entire installation has successfully completed.
- kCancel — Sent after an installation was stopped due to an error or cancelation by the user, Action Atom or Atom Extender.

The Atom Extender must return a result telling Installer Engine what to do after receiving each message. The result codes, kFatalError, kContinueAsNormal, kCancelInstallation can be returned for any message. If you do not want to change the default action of Installer Engine, return kContinueAsNormal; otherwise, return kFatalError or kCancelInstallation to stop the installation. If the message was kBeforePart and you've copied the data yourself (with or without the supplied Installer routines) then return kCopiedData.

Memory Allocation within Atom Extenders

When the Atom Extender is executed, memory allocated using Macintosh OS/Toolbox routines will come from a sub-heap created by Installer Engine. The size of the sub-heap is determined by the requested free memory field of the 'inex' resource, and the available room in Installer Engine's heap. You must check the actual size of the heap to make sure you were allocated all of the memory you requested, since Installer Engine calls you whether or not all your request was allocated. This helps Installer Engine maintain better control over all memory allocation.

The scriptwriter may need to adjust the size of Installer Engine's partition to make sure enough memory is always available for the Atom Extender's needs. The following equation helps determine Installer Engine's partition size for the required free memory and other factors.

> runTimeSize = size of the largest: 'FOND' resource, Desk Accessory resource, or owned resource being copied.
>
> extenderSize = largest required free memory of any Atom Extender + code size of all Atom Extenders, in use during an installation + the total amount of static memory allocated.
>
> scriptSize = the uncompressed size of the script
>
> estimated partition size = (the larger of: runTimeSize, extenderSize) + scriptSize + 350K

Installer Engine provides memory management routines of its own to allow Atom Extenders to use available memory from the MultiFinder and Installer heaps. These routines are very similar to the Memory Manager calls, but are needed to maintain compatibility with pre-7.0 MultiFinder temporary memory calls.

- INewHandle — Similar to NewHandle, but always use IDisposHandle to dispose this handle, and IHLock and IHUnlock to lock and unlock this handle. The handle may or may not have been allocated in the MultiFinder heap.
- IDisposHandle — Similar to DisposHandle.
- IHLock — Similar to HLock.
- IHUnlock — Similar to HUnlock.

The INewHandle routine has been provided for two main purposes. The first purpose is to allow access to extra memory, when available, to make the actions of an Atom Extender more

efficient. The second purpose is to allow memory to be maintained over invocations of an Atom Extender.

Running within a Sub-Heap

A sub-heap for memory allocation is mandatory for Atom Extenders to prevent gridlock when buffering data. Sub-heaps are optional for the other code resources that can be called during an installation, such as search procedures, Action Atoms and rule functions. We suggest some do's and don'ts on dealing with sub-heaps below.

DO's:

- Sub-heaps are mandatory for Atom Extenders, but other code resources can run inside Installer Engine's heap, so enter 0 (size) as the request memory size if you don't want to bother with sub-heaps.
- Set the current zone to the application heap before calling any Toolbox routine, and restore it afterwards. This assures that memory allocated within these calls won't get left in your sub-heap.
- Use Installer Engine's memory routines, unless you want the memory allocated directly from your sub-heap.

DON'Ts:

- Watch out for memory that is left in the sub-heap after you return control to Installer Engine that the system may have references to. If you set the current zone to the application heap before calling any Toolbox routine you shouldn't have any problems. For example, if you call the GetResource routine (without properly setting the zone) that allocates a handle inside your sub-heap Installer Engine may crash after returning control to Installer Engine
- Never return a reference to a handle allocated in your sub-heap for the `fStaticDataHdl` field. Since the sub-heap is destroyed between invocations of the code resource, it's best to use `INewHandle` to allocate any handle you store in the `fStaticDataHdl` field.
- Never assume you received a sub-heap of the full size requested. Check that you have enough memory enough to complete your task, since Installer Engine calls you whether or not your full request was allocated. Look for warning messages in Installer Debugger to help identify this situation.

Converting Existing Decompression Code

Conversion is simple for those decompressors that access files using standard File Manager calls. Installer Engine routines have been designed to easily replace calls to the high-level File Manager routines: FSRead, FSWrite, SetFPos, GetFPos and GetEOF.

If your decompression code must use a temporary file to store partially decompressed data, then it's up to you to create and destroy this file. Space on the target disk can be reserved by adding the needed space to one File Atom installed to the target disk. As an alternative, Installer Engine `INewHandle` routine can be used to allocate a temporary buffer if sufficient memory exists, but the availability of this memory is never guaranteed.

If your original decompression code comes from an application and you are using the MPW development system, make sure to read Tech. Note #256. To access global data you will need to create your own A5 world and switch back to Installer Engine's A5 world before calling any

Installer routine or Mac Toolbox trap. Avoiding global data makes writing code resources easier unless your development system (i.e. THINK C) supports using the A4 register to access global data.

Atom Extender Reference

This section describes the data structures and routines that are used by Atom Extenders. You'll need to include the interface file "AtomExtenderHeader.h" or "AtomExtenderHeader.p" to have access to these data structures and routines.

Resource Description

Information about how to call the Atom Extender code resource is contained in the script resource of type 'inex'. New versions of File Atoms, Resource Atoms and Font Atoms reference the 'inex' resource using its resource ID. The resource has the template:

```
#define atomExtenderFlags                                       \
    boolean    dontSendInitMessage, sendInitMessage;            \
    boolean    dontSendBeforeMessage, sendBeforeMessage;        \
    boolean    dontSendAfterMessage, sendAfterMessage;          \
    boolean    dontSendSuccessMessage, sendSuccessMessage;      \
    boolean    dontSendCancelMessage, sendCancelMessage;        \
    boolean    continueBusyCursors, suspendBusyCursors;         \
    fill bit[10];

type 'inex' {
       switch {
           case format0:
               key integer = 0;         /* Extender Format version */
               atomExtenderFlags;       /* Flags for Format 0 */
               unsigned longInt;        /* Code resource Type */
               integer                  /* Code resource ID */
               longInt;                 /* Refcon Value */
               longInt;                 /* Required Free Memory */
               evenPaddedString;        /* Status Description */
       };
};
```

Flag descriptions

`sendInitMessage` — Asks to send an `kInitialize` message to those Atom Extenders attached to atoms that will be executed during the installation. The Atom Extender receives the `kInitialize` message after each time the user clicks the Install button and preflighting is successful. The purpose of the `kInitialize` message is to allocate any static memory needed to communicate between Atom Extenders.

`sendBeforeMessage` — Asks to send a `kBeforePart` message to the Atom Extender code resource when the atom part is ready to be copied. Installer Engine has found and opened the source file that the part exists in and is ready to begin buffering the source data. The Atom Extender can override Installer Engine's default copy mechanism by using the supplied routines and returning a `kCopiedData` result code.

sendAfterMessage	Asks to send an kAfterPart message to the Atom Extender code resource for each part that has been successfully written by Installer Engine or Atom Extender. An kAfterPart message is only sent if Installer Engine performed the copy or the Atom Extender called WriteTargetData (with data length greater than 0) during a kBeforePart message.
sendSuccessMessage	Asks to send a kSuccess message to the Atom Extender code resource after the entire installation has successfully completed. The purpose of the kSuccess message is to allow disposal of any static memory allocated during the kInitialize message, and delete any temporary files that were created by the Extender. Only those Atom Extenders that received kInitialize messages will be sent a kSuccess message.
sendCancelMessage	Asks to send a kCancel message to the Atom Extender code resource when the user cancels the installation, an error forces the installation to stop, or an Atom Extender cancels the installation. The purpose of the kCancel message is to reverse any changes that were made during the installation, and dispose of any static memory allocated during the kInitialize message. Only those Atom Extenders that received kInitialize messages will be sent a kCancel message.

Field descriptions

Code Resource Type	The resource type of the Atom Extender code resource to be called. (4-bytes)
Code Resource ID	The resource ID of the Atom Extender code resource to be called. (2-bytes)
RefCon Value	The value that will be passed in the fRefCon field of the Atom Extender parameter block. (4-bytes)
Required Free Memory	The minimum number of free bytes the Atom Extender needs in its own heap to make local allocation of memory using Macintosh OS/Toolbox routines. (4-bytes)
Status Description	An optional string that is displayed in the Status dialog during execution of the Atom Extender. (even-padded Pascal string)

Data Structures

Function Interface

The entry point of the Atom Extender code resource must have the interface:

```
ResultCode MyAtomExtender(ExtenderPBPtr myExtenderPBPtr);
```

The Atom Extender should return a result telling Installer Engine what to do after executing the code resource. You can use these constants to specify the result code:

```
typedef enum {
    kFatalError = -1,            /* Stop installation with error user dialog. */
    kContinueAsNormal = 0,       /* Request default action from Installer. */
    kCancelInstallation = 1,     /* Cancel installation with cancel user dialog. */
    kCopiedData = 2              /* Override default copy on kBeforePart message. */
} ResultCode;
```

Parameter Block

The AtomExtender parameter block contains information about the atom being installed. Most Atom Extenders will only need to reference a few fields from this parameter block. The parameter block has the structure:

```
typedef  struct {
    ExtenderMessageID fMessageID;
    Handle            fStaticDataHdl;
    ProcPtr           fCallBackProcPtr;
    short             fTargetVRefNum;
    long              fTargetFolderDirID;
    short             fSystemVRefNum;
    long              fSystemBlessedDirID;
    long              fRefCon;
    DataType          fDataType;
} EnvironsHeaderRec;

typedef  struct {
    FSSpec            fSourceFile;
    FSSpec            fTargetFile;
    long              fTotalTargetSize;
    long              fTargetPosStart;
    long              fTargetPartSize;
    long              fSourcePartSize;
    long              fInstallerTempDirID;
} CopyHeaderRec;

typedef  struct {
    EnvironsHeaderRec     fEnvironmentHeader;
    CopyHeaderRec         fCopyPBHeader;
} BasicCopyRec;

typedef  struct {
    EnvironsHeaderRec     fEnvironmentHeader;
    CopyHeaderRec         fCopyPBHeader;
    ResType               fSourceRsrcType;
    short                 fSourceRsrcID;
    ResType               fTargetRsrcType;
    short                 fTargetRsrcID;
    Str255                fTargetRsrcName;
    short                 fTargetRsrcAttrs;
} RsrcCopyRec;

typedef  struct {
    EnvironsHeaderRec     fEnvironmentHeader;
    TCopyHeaderRec        fCopyPBHeader;
    ResType               fSourceFontRsrcType;  /* usually 'part' */
    short                 fSourceFontRsrcID;
    ResType               fTargetFontRsrcType;  /* FONT, NFNT, or sfnt */
    short                 fTargetFontRsrcID;
    Str255                fTargetFontRsrcName;
    short                 fTargetFontRsrcAttrs;
    Str255                fFamilyName;
    short                 fFamilyID;
```

```
    short               fFontSize;
    short               fFontStyle;
} FontCopyRec;

typedef union {
    BasicCopyRec        fBasicPBRec;
    BasicCopyRec        fFileCopyPBRec;
    RsrcCopyRec         fRsrcCopyPBRec;
    FontCopyRec         fFontCopyPBRec;
} ExtenderPBRec, *ExtenderPBPtr;
```

Field descriptions

`fMessageID` One of five messages the Atom Extender will receive when being called. Use these constants to understand the message ID:

```
typedef enum {
   kInitialize = 0,
   kBeforePart = 1,
   kAfterPart  = 2,
   kSuccess    = 3,
   kCancel     = 4
} ExtenderMessageID;
```

Certain fields in the Atom Extender parameter block may or may not be valid when receiving a specific message. Each of the following field descriptions include a note about when the field is valid.

`fStaticDataHdl` A handle created by the Atom Extender using INewHandle to save information between calls to this Atom Extender code resource. The field is always NULL when receiving an `kInitialize` message. You can assign a value to this field during any call to the Atom Extender, and receive the same value in this field during the next message. (Valid for all messages.)

`fCallBackProcPtr` A pointer to Installer Engine's dispatcher routine. You'll need to pass this field as a parameter to glue routines that provide access to Installer functions. (Valid for all messages.)

`fTargetVRefNum` The target disk's vRefNum. When allowing the user to select a target application folder, this is the volume on which the folder resides. (Valid for all messages.)

`fTargetFolderDirID` The target application folder's directory ID. This value is -1 if the user cannot select a target application folder. Target File Specs that use the reserved folder path `folder-user` will be placed in this folder. (Valid for all messages.)

`fSystemVRefNum` The System disk's vRefNum. Target File Specs that use the reserved folder path `special-xxx` will be placed in the System Folder on this volume. (Valid for all messages.)

`fSystemFolderDirID` The directory ID of the System Folder on the disk with the refNum `fSystemVRefNum`. This directory is not

necessarily the currently active System Folder. (Valid for all messages.)

`fRefCon` — A 4-byte value defined by the scriptwriter in the 'inex' script resource. (Valid for all messages.)

`fDataType` — One of five values that specifies the type of data the Atom Extender has been given. Use this value to determine which type of atom is being installed, and which variant of the parameter block should be used. Use these constants to understand the data type:

```
typedef enum {
   kFileAtomDataFork              = 0,
   kFileAtomRsrcFork              = 1,
   kFileAtomRsrcForkInDataFork    = 2,
   kRsrcAtom                      = 3,
   kFontAtom                      = 4
} DataType;
```

(Valid for all messages.)

`fSourceFile` — An FSSpec record specifying the source file from which the source data is read. (Valid only for `kBeforePart` and `kAfterPart` messages.)

`fTargetFile` — An FSSpec record specifying the target file to which the target data is written. (Valid only for `kBeforePart` and `kAfterPart` messages.)

`fTotalTargetSize` — The number of bytes all source pieces will occupy in the target. (Valid only for `kBeforePart` and `kAfterPart` messages.)

`fTargetPosStart` — The offset in bytes into the target data the part's data is written. (Valid only for the `kBeforePart` and `kAfterPart` messages.)

`fTargetPartSize` — The size in bytes the part's data will occupy in the target. (Valid only for the `kBeforePart` and `kAfterPart` messages.)

`fSourcePartSize` — The size in bytes the part's data occupies in the source. (Valid only for the `kBeforePart` message.)

`fInstallerTempDirID` — The directory ID of the Installer temp folder on the target's volume that contains the original target file. If this field is -1, the original file was not saved or did not exist. You can get the target's volume refNum from the `fTargetFile` field. (Valid only for the `kBeforePart` and `kAfterPart` messages.)

Field descriptions for Resource Atoms

`fSourceRsrcType` — The resource type of the source resource part. (Valid only for `kBeforePart` and `kAfterPart` messages.)

`fSourceRsrcID` — The resource ID of the source resource part. (Valid only for `kBeforePart` and `kAfterPart` messages.)

`fTargetRsrcType`	The resource type of the target resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fTargetRsrcID`	The resource ID of the target resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fTargetRsrcName`	The resource name to be given to the target resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fTargetRsrcAttrs`	The resource attributes to be given to the target resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)

Field descriptions for Font Atoms

`fSourceFontRsrcType`	The resource type of the source font resource part. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fSourceFontRsrcID`	The resource ID of the source font resource part. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fTargetFontRsrcType`	The resource type of the target font resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fTargetFontRsrcID`	The resource ID of the target font resource, which is not known until the part has been written by Installer Engine. (Valid only for the `kAfterPart` messages.)
`fTargetFontRsrcName`	The resource name to be given to the target font resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fTargetFontRsrcAttrs`	The resource attributes to be given to the target font resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fFamilyName`	The family name (i.e. “Times”, “Geneva”…). The source ‘FOND’ resource can be found using this name. (Valid only for `kBeforePart` and `kAfterPart` messages.)
`fFamilyID`	The family ID. Normally this is the resource ID of the ‘FOND’ resource. (Valid only for `kBeforePart` and `kAfterPart` messages.)

Using Version Compare Functions (‘invc’)

Version Compare functions allow external code resources to be called when the target file or resource’s version must be determined in order to compare with the source version number supplied in the script.

The Installer 3.X versions allow the scriptwriter to specify how it should handle the case where the target file is newer than the source file. This usually prevents downgrading the user’s software, but the opposite case may occur where downgrading is required to maintain sync-ed versions of software. Previously, the determination of “newness” was dependent on comparing the creation dates of the target file to the value in the source ‘infs’ resource. This works most of the time, but is not a reliable way to compare files.

All files being installed should have a version resource (‘vers’ 1 and maybe a ‘vers’ 2), and the ability to “newness” of files based on version resource is a desirable feature. Installer Engine provides three ways to compare files:

- Old way using creation date. Use the `useSrcCrDateToCompare` flag in the File Atom.
- Default version number comparison using the ‘vers’ 1 resource in the target file. Use the `useVersProcToCompare` flag in the File Atom. Place the source version number in the `Source Version Number` field and place 0 (zero) in the `Version Compare Rsrc ID` field of the File Atom.
- Custom determination of the target version number of a file or resource using a code resource. Use the `useVersProcToCompare` flag in the File Atom. Place the source version number in the `Source Version Number` field and place the ‘invc’ resource ID in the `Version Compare Rsrc ID` field of the File Atom or Resource Atom.

Using Version Compare Functions with File Atoms

During preflighting of the target disk Installer Engine will call the Version Compare code resource for each existing target file that references a valid ‘invc’ script resource. Upon entering the code resource the current resource file will have been set to the target file, so the code resource can easily make Resource Manager calls to read an alternative version resource. If the version information is stored in the data fork, or in another file the code resource must open and close the file themselves.

The Compare Version code resource must return the version number as the function result. If an error occurs while finding the version number, then return 0 (zero).

Allowable Installer functions when comparing the version of a file:

- Memory functions: INewHandle, IDisposHandle, ILockHandle, IUnLockHandle.
- Action Handler function: RegisterAction.

Using Version Compare Functions with Resource Atoms

During preflighting of the target disk Installer Engine will call the Version Compare code resource for each existing target resource that references a valid ‘invc’ script resource. Upon entering the code resource the current resource file will have been set to the file the target resource resides. The code resource should call Installer Engine’s `ReadTargetData` routine to read from the target resource. This should allow most resource version schemes using a header to easily and quickly determine the version number of the resource. Using Resource Manager calls should be avoided.

The Compare Version code resource must return the version number as the function result. If an error occurs while finding the version number, then return 0 (zero).

Allowable Installer functions when comparing the version of a resource:

- Memory functions: INewHandle, IDisposHandle, ILockHandle, IUnLockHandle.
- Target functions: ReadTargeData, GetTargeDataEOF, GetTargeDataPos, SetTargeDataPos.
- Action Handler function: RegisterAction.

Version Compare Runtime Environment

The Version Compare code resource shares memory space with Installer Engine. Local memory allocation using Memory Manager calls will come from Installer Engine's heap. Please be nice to our heap.

Version Compare Function Reference

This section describes the function interface and resource descriptions needed to use the Version Compare script resource.

Function Interface

Installer Engine calls your Version Compare code resource assuming the following function interface.

```
long ComputeVersionNumber( ComputeVersionPBPtr );
```

The Version Compare function writer must return a version number. The version number consists of four bytes defined from the high byte as: major revision level, minor revision level, development stage, and prerelease revision level. See *Inside Macintosh: Macintosh Toolbox Essentials*, page 7-31 for more details.

Parameter Block

The version compare code resource is passed a pointer to a parameter block containing the callback pointer.

```
typedef  struct {
    ProcPtr       fCallBackProcPtr;
} ComputeVersionPB, *ComputeVersionPBPtr;
```

Field descriptions

`fCallBackProcPtr` A pointer to Installer Engine's dispatch routine. You'll need to pass this field as a parameter to Installer function glue routines.

Resource Description

The template of the Version Compare script resource is shown below.

```
#define versionCompareFlags      \
       fill bit[16];

type 'invc' {
       switch {
           case format0:
               key integer = 0;      /* Format version */
               versionCompareFlags;  /* Version Compare */
               literal longint;      /* Version Compare Code Rsrc Type */
               integer;                  /* Version Compare Code Rsrc ID */
               longint;                  /* Minimal Requested Memory */
               evenPaddedString;     /* Summary */
       };
};
```

Field descriptions

`Version Compare Flags` Currently reserved for use by Apple Computer, Inc. (2-bytes)

`Version Compare Code Rsrc Type`
The resource type of the Version Compare code resource. (4-bytes)

`Version Compare Code Rsrc ID`
The resource ID of the Version Compare code resource. (2-bytes)

`Minimal Required Memory` The minimum number of free bytes the Version Compare code resource needs during execution. (4-bytes)

NOTE
Unless your Version Compare routine has a special need for it's own memory heap, you should always assign this field a value of zero. A value of zero allows the Version Compare routine to use Installer Engine's allocated memory. ◆

`Version Compare Summary` An optional string briefly describing the purpose of this Version Compare function. This string is never displayed to the user. (even-padded Pascal string)

CHAPTER 4

File Specification

This chapter describes how to create the script resources that specify the source and target file.

About File Specifications

Most atoms require a source and target file to be specified in order to carry out the action defined by the atom. For example, when using a File Atom to copy a file from the source disk to the target disk, a Target File Spec resource (‘intf’) and a Source File Spec resource (‘infs’) must be referenced from the atom.

Example (Part 3): Specifying Target and Source Files

During part 2 of this example, we created a ‘ifa#’ resource that contains seven File Atoms to copy our seven files. Now we need to specific where the files reside on the source disk, and location we wish to copy the files to on the target disk.

Step 9: Create the Target File Specifications

We’ll store our target file specifications inside a ‘itf#’ resource. Each target file specification will have a record ID that corresponds to the value we store in each File Atom. The file paths are stored in a separate resource of type ‘ist#’. Both resources are defined below:

```
resource 'itf#' (300) {
    format0 {
        300,       // ID of 'ist#' resource containing path names.
        {
            /* [1] */  1001,                // Unique ID of this target file specification
                       noSearchForFile,     // We don't need to search for the target file
                       TypeCrNeedNotMatch,  // We don't care what the existing type and creator are
                       'ttro',              // Type given to the file
                       'ttxt',              // Creator given to the file
                       0x0,                 // Finder flags given to file, filled in by ScriptCheck
                       0x1,                 // Creation date given to file, value of 1 specifies
                                            // that ScriptCheck should update this value
                       0x1,                 // Modification date given to file, value of 1 specifies
                                            // that ScriptCheck should update this value
                       0,                   // No search proc specified since we're not searching
                       1,                   // Index of the file path in the 'ist#' resource
            /* [2] */  1002, noSearchForFile, TypeCrNeedNotMatch, 'ttro', 'ttxt', 0x0, 0x1, 0x1, 0, 2,
```

```
            /* [3] */  2001, noSearchForFile, TypeCrNeedNotMatch, 'ttro', 'ttxt', 0x0, 0x1, 0x1, 0, 3,
            /* [4] */  3001, noSearchForFile, TypeCrNeedNotMatch, 'ttro', 'ttxt', 0x0, 0x1, 0x1, 0, 4,
            /* [5] */  3002, noSearchForFile, TypeCrNeedNotMatch, 'ttro', 'ttxt', 0x0, 0x1, 0x1, 0, 5,
            /* [6] */  1004, noSearchForFile, TypeCrNeedNotMatch, 'ttro', 'ttxt', 0x0, 0x1, 0x1, 0, 6,
            /* [7] */  1005, noSearchForFile, TypeCrNeedNotMatch, 'ttro', 'ttxt', 0x0, 0x1, 0x1, 0, 7,
        }
    }
};

resource 'ist#' (300) {
    format0 {
        {
            /* [1] */  ":User Interface Example:Example File • 1",
            /* [2] */  ":User Interface Example:Example File • 2",
            /* [3] */  ":User Interface Example:Example File • 3",
            /* [4] */  ":User Interface Example:Example File • 4",
            /* [5] */  ":User Interface Example:Example File • 5",
            /* [6] */  ":User Interface Example:Example File • 6",
            /* [7] */  ":User Interface Example:Example File • 7",
        }
    }
};
```

Step 10: Create the Source File Specifications

For the source file specifications, we must create a separate ‘infs’ resource for each of the seven files.

```
resource 'infs' (1001) {
    'ttro',                         // Type for source file
    'ttxt',                         // Creator for source file
    0x1,                            // Creation date for source file, value of 1 specifies
                                    // that ScriptCheck should update this value
    noSearchForFile,                // ignored
    TypeCrMustMatch,                // Type & Creator must match file on source disk
    "Example Files:Example File • 1" // Path to source file
};

resource 'infs' (1002) { 'ttro', 'ttxt', 0x1, noSearchForFile, TypeCrMustMatch,
                         "Example Files:Example File • 2" };

resource 'infs' (2001) { 'ttro', 'ttxt', 0x1, noSearchForFile, TypeCrMustMatch,
                         "Example Files:Example File • 3" };

resource 'infs' (3001) { 'ttro', 'ttxt', 0x1, noSearchForFile, TypeCrMustMatch,
                         "Example Files:Example File • 4" };

resource 'infs' (3002) { 'ttro', 'ttxt', 0x1, noSearchForFile, TypeCrMustMatch,
                         "Example Files:Example File • 5" };

resource 'infs' (1004) { 'ttro', 'ttxt', 0x1, noSearchForFile, TypeCrMustMatch,
                         "Example Files:Example File • 6" };

resource 'infs' (1005) { 'ttro', 'ttxt', 0x1, noSearchForFile, TypeCrMustMatch,
                         "Example Files:Example File • 7" };
```

Specifying Target Files (‘intf’ & ‘itf#’)

The goal of the Target File Spec is to specify where to install the file on the target disk. In most cases the scriptwriter can hard code the path to the target file using the various reserved folder path names, or in special cases, search for the file using a code resource.

The Target File Spec can be stored in one of two resources types. The older ‘intf’ resource requires a separate resource for each target file, but the newer ‘itf#’ stores all file specs in a single resource. Since you lose no functionality by using the new format, we recommend you store target file specs in a ‘itf#’ resource.

The path to the target file that must start with a colon or one of two reserved tokens (special-xxxx or folder-user). Because the target volume name is not known at the time a script is written, the scriptwriter supplies a partial path name that begins in the root directory of the target disk.

Storing Target File Specs in the ‘ift# Resource

When referencing a target file spec from an atom, Installer Engine will look first for a target file spec with the specified record ID in the ‘itf#’ resource, and if not found will look for a ‘inft’ resource with the specified ID. NOTE: an Installer script can only have one ‘itf#’ resource.

Installing into Special Folders

A scheme is provided to support special folders such as the System Folder, the Apple Menu Folder, and other Folder Manager folders. The short-hand notation used to specify a folder manager folder is “`special-xxxx`”, where the “xxxx” is one of the defined special folder types. The special folder types currently defined for System 7.0 through 8.5 are as follows:

Type	Folder	Type	Folder
`'macs'`	System Folder	`'prnt'`	PrintMonitor Documents
`'strt'`	Startup Items	`'amnu'`	Apple Menu Items
`'extn'`	Extensions	`'pref'`	Preferences
`'ctrl'`	Control Panels	`'font'`	Fonts (System 7.1 only)
`'shdf'`	Shutdown Items	`'macD'`	System Folder (Disabled)
`'ctrD'`	Control Panels (Disabled)	`'extD'`	Extensions (Disabled)
`'strD'`	Startup Folder(Disabled)	`'shdD'`	Shutdown Folder(Disabled)
`'amnD'`	Apple Menu (Disabled)	`'issf'`	Internet Search Sites
`'laun'`	Launcher Items	`'fnds'`	Find
`'fbcf'`	TheFindByContentFolder	`'ilgf'`	Installer Logs
`'root'`	target volume root	`'desk'`	Desktop Folder
`'trsh'`	Trash	`'empt'`	Network Trash Folder
`'temp'`	Temporary Items	`'flnt'`	Cleanup At Startup
`'asup'`	Application Support	`'apps'`	Applications
`'docs`	Documents	`'odod'`	OpenDoc
`'odsp'`	OpenDoc Shell Plug-Ins	`'odlb'`	OpenDoc Libraries
`'oded'`	Editors	`'ƒtex'`	Text Encodings
`'odst'`	Stationery	`'ƒhlp'`	Help
`'ƒnet'`	Internet Plug-Ins	`'ƒmod'`	Modem Scripts
`'ppdf'`	Printer Descriptions	`'ƒprd'`	Printer Drivers
`'ƒscr'`	Scripting Additions	`'ƒlib'`	Extensions (for shared libraries)
`'fvoc'`	Voices	`'sdev'`	Control Strip Modules
`'astƒ'`	Assistants	`'utiƒ'`	Utilities
`'aexƒ'`	Apple Extras	`'cmnu'`	Contextual Menu Items
`'morƒ'`	Mac OS Read Me Files	`'prof'`	ColorSync Profiles
`'appr'`	Appearance	`'snds'`	Sound Sets
`'thme'`	Theme Files	`'dtpƒ'`	Desktop Pictures
`'favs'`	Favorites	`'fasf'`	Folder Action Scripts
`'scrƒ'`	Scripts	`'astƒ'`	Assistants
`'rapp'`	Recent Applications	`'rdoc'`	Recent Documents

'rsvr'	Recent Servers	'spki'	Speakable Items
'intf'	Internet	'amnD'	Apple Menu (Disabled)

This means that if the fully qualified path to the system file on the target disk was "MyVolume:MyFolder:MySystemFolder:System", then a reference to the special folder "special-macs" would be expanded by Installer Engine to be "MyVolume:MyFolder:MySystemFolder." For example, if you wanted to install the Finder file into the blessed folder on the system disk, you would use the path name "special-macs:Finder" in the File Spec. If a folder in the path name does not exist, Installer Engine creates it. Installer Engine supports the use of special folders even when running on systems which do not have Folder Manager functionality. For more information about the Folder Manager, refer to the Finder chapter in *Inside Macintosh, Volume VI.*

To support installing into pre-7.0 systems, Installer Engine will automatically map special folders to the System Folder. For example, if your target is the Extensions Folder then Installer Engine will change the destination to the System Folder rather than create an Extensions Folder. This also applies to rule clauses that reference a file spec. While this simplifies script writing for many files that go into special folders, it does not simplify DA installation.

In addition to the folder types defined by Apple, Installer Engine supports five special autorouting folder types that are only valid when used with "special-" within target file specs. The auto routing feature allows the scriptwriter to search for the existence of a file in either the main folder or the disabled folder. If a file is found in any of these folders, the same folder will be used for installation. If the file is not found, the new file will be installed by default in the enabled folder. If a file is found in both the enabled and disabled folder, the new file will replace the file in the enabled folder. These autorouting folder types are as follows:

Type	Folder	Type	Folder
'macX'	auto routing to System Folder	'amnX'	auto routing to Apple Menu Items
'ctrX'	auto routing to Control Panels	'strX'	auto routing to Startup Folder
'shdX'	auto routing to Shutdown Folder		

Localized systems are supported through a defined algorithm for determining the correct special folder name by searching for the proper 'fld#' or 'nfd#' resource. Given a special folder type, Installer Engine searches for the type in a 'fld#' or 'nfd#' resource in the following order:

- Target System file
- Installer Script file
- Installer Engine application file
- Booted System file

Generally, the scriptwriter should not include a 'fld#' or 'nfd#' resource in the script so installations into localized System Folders will happen correctly.

Installing into the User-Selected Application Folder

The scriptwriter can easily install or remove files and resources from the application folder selected by the user. This reserved folder name should only be used in conjunction with the application folder interface mode. For example, to install the file "Demo Application" into the application folder the path name might be "folder-user:Demo Application".

Managing Rollbacks on Multiple Target Volumes

With the ability to install onto more than one target volume at a time, a complete rollback is not possible unless the original files on all target volumes are maintained. This is not always

possible due to available disk space on the target volumes. Installer Engine always saves the original System Folder files on the boot volume, but makes the saving of all other original files optional based on the amount of available disk space on each target volume. If there is enough space to save all original files on all target volumes the 'Cancel' button shows in the Status dialog thereby allow for a complete rollback, otherwise a 'Stop' button is shown.

Installing onto the Installer Volume

To support installation onto the Installer volume, Installer Engine must prevent source files from being disturbed. After the user clicks the Install button, Installer Engine determines whether any target file matches the location of a source file. If there is a conflict the user is alerted. This case is much more likely to happen when the scriptwriter allows the user to select the target application folder.

Setting the Finder flags and Dates

Format 1 of the 'intf' resource supports supplying the Finder flags, creation date and modification date to make target file specification of archived files easier. Installer Engine sets the values of a newly created file to the specified values in the 'intf'. If the 'intf' is referenced from a File or ResMerge Atom and the file already exists the Finder flags, creation and modification dates will be updated. The Finder Flags in a File Atom will always override the Finder Flags specified in the 'intf' resource.

If you use an 'infs' or format 0 of the 'intf' Installer Engine continues to use the Finder flags, creation and modification date of the source file. In cases where the source file is a compressed archive, format1 version of the 'intf' is recommended. ScriptCheck will fill in the correct flags and dates if the file does not reference an Atom Extender. If an Atom Extender is being used, make sure to supply the appropriate ScriptCheck extension. To have ScriptCheck fill in the correct values, place a zero in the Finder flags field and 1 in the Creation Date and Modification Date fields. If the date field is zero or 1, Installer Engine reverts to the older method of determining the correct target date, such as with format 0 of the 'intf'.

Specifying Source Files ('infs')

The goal of the Source File Spec is to describe where the file resides on the original distribution source disk.

NOTE

When performing installations from some network servers, it is necessary that all source files have a file creation and modification time stamp that has an even value for the seconds. If you intend for installation to be performed from network servers you should set the creation and modification times of all your source files to a value such as 12:00:00 PM. ◆

Source Disk Search Path

Because of multiple disk installation capability, the semantics of the File Name field are a bit complex. Customers may have copied the distribution disks onto a file server and want to install from there, or they may have made floppy backup copies of the system disks provided by Apple and expect to be able to install from those. They might also want to set up a server that users can install from. In either case, the path name specified in the File Name field of the Source File Spec will probably be slightly off (e.g. the backup disks may not have the same names as the originals) so we need to be somewhat flexible when searching for a specific source

file. In a Source File Spec, the full path name (including volume name) must be given in the File Name field, but it may not be used by Installer Engine exactly as given.

While Installer Engine is flexible as to exact volume names, it requires that disk contents be the same as what is specified in the script. Installer Engine uses the scripts to make a list of the volumes that are needed for the installation, and the files that should be on each volume. All files that are supposed to be on the same volume must be on the same volume, but that volume's name may be different from the name given in the script. In addition, we relax this condition slightly to allow for hard disk and network installations. When Installer Engine needs a disk specified in the script, it searches in the following order:

- Search for a folder with the disk's name at the same HFS level as the script.
- Search for a folder with the disk's name at the script parent's HFS level.
- Search for a folder with the disk's name at the same HFS level as Installer Engine.
- Search for a folder with the disk's name at Installer Engine parent's HFS level
- Search for an online volume with the correct name.
- If Installer Engine was not launched from an AppleShare server volume, ask for a floppy disk or CD-ROM.

For backup purposes, Installer Engine makes the assumption that source diskettes may have the disk name changed. For example, the user may omit such special characters such as '©' and '™' when making the backup copies of the original installations disk. To find source files, Installer Engine parses off the disk name and searches for the required files from the root level of the current diskette. If the file is not found, the search criteria listed above are used.

The disk that Installer Engine is on is a special case in our requesting disk and disk content strategies. If possible, Installer Engine's disk is always the first disk that we copy files and resources from. Also, the contents of Installer Engine disk are checked for correctness before anything is deleted or copied. We recommend that scriptwriters (especially for system releases) have the most important files and resources to be copied on Installer Engine disk, since that's the only disk that we can guarantee the user has. Otherwise, Installer Engine could start the installation, which includes deleting all appropriate target files and resources, and have to ask for a disk that contains a crucial file, in which case the user cancels out of the dialog, since that disk is for some reason not available, thereby aborting the installation, leaving the user worse off than before (i.e., a system that no longer boots). This, of course, does not happen when doing a live install

File spec. IDs of 0 should only be used for source files specifications for the deletion of files; when the `noCopy` and `deleteOnInstall` flags are set in the `'infs'` resource

File Spec. Reference

This section describes the data structures and script resources needed to reference source and target files from atoms.

Resource Descriptions

The section describes the resource definitions for the target file specification resources ('intf', 'itf#', and 'ist#') and the source file specification resource ('infs').

Target File Spec. Resource (‘intf’)

The Target File Spec. resource contains a reference to an ‘insp’ script resource which describes the file searching code resource, much like the Atom Extender script resource does.

```
#define targetFileFlags                                          \
        boolean    noSearchForFile, SearchForFile;         \
        boolean    TypeCrNeedNotMatch, TypeCrMustMatch;    \
        fill bit[14]                    /* Reserved    */

type   'intf' {
    switch {
        case format0:
            key integer = 0;         /* Target File Spec. Format version */
            fileSpecFlags;           /* Target File Spec. Flags */
            literal longint;         /* Target File Type */
            literal longint;         /* Target File Creator */
            literal integer;         /* Search Proc. Rsrc ID */
            evenPaddedString;        /* Target File Path */
        case format1:
            key integer = 1;         /* Target File Spec. Format version */
            fileSpecFlags;           /* Target File Spec. Flags */
            OSType;                  /* Target File Type */
            OSType;                  /* Target File Creator */
            unsigned hex integer; /* Target File Finder Flags */
            unsigned hex longint; /* Target File Creation Date */
            unsigned hex longint; /* Target File Mod. Date */
            rsrcID;                  /* Search Proc. Rsrc ID */
            evenPaddedString;        /* Target File Path */
    };
};
```

Flag descriptions

`noSearchForFile/SearchForFile`
: Determines if the Search Procedure code resource will be called. Use the `searchForFile` flag to override the specified file path and use the list of files returned from the Search Procedure code resource.

`TypeCrNeedNotMatch/TypeCrMustMatch`
: Determines if the specified type and creator must match the file found on the target disk. If the type and creator of the found file do not match those specified, the installation stops the installation. This is also true of any found files returned by the Search Procedure.

Field descriptions

`Target File Type`
: The file’s type if Installer Engine must create one during a copy. If the `TypeCrMustMatch` flag is used and an existing file is found, its type must match the type entered in this field. (4-bytes)

`Target File Creator`
: The file’s creator if Installer Engine must create one during a copy. If the `TypeCrMustMatch` flag is used and an existing file is found, its creator must match the creator entered in this field. (4-bytes)

`Target Finder Flags`
: The Finder flags to be given to a new file. If referenced from a ResMerge Atom, the Finder Flags are given to the existing

file. The Finder flags specified in a File Atom always override this field. Only available in format 1. (2-bytes)

`Target Creation Date` The creation date to be given to a new file. If referenced from a File Atom or ResMerge Atom, the existing file is given this creation date. Leave as 0 to use the source file's or today's date. Set to 1 to have ScriptCheck update using the source file's creation date. Only available in format 1. (4-bytes)

`Target Mod. Date` The modification date to be given to a new file. If referenced from a File Atom or ResMerge Atom, the existing file is given this modification date. Leave as 0 to use the source file's or today's date. Set to 1 to have ScriptCheck update using the source file's modification date. Only available in format 1. (4-bytes)

`Search Procedure Rsrc ID` The resource ID of an 'insp' resource, which describes the code resource to call to perform the target file search. (2-bytes)

`Target File Path` The partial path to the target file that must start with a colon or one of two reserved tokens (special-xxxx or folder-user). (even-padded Pascal string)

Target File Spec List Resources ('itf#' & 'ist#')

```
type 'itf#' {
       switch {
           case format0:
               key integer = 0;                 /* Target File Spec. Format version */
               rsrcID;                          /* File Paths Rsrc ID */
               integer = $$Countof (TargetRec); /* Number of records */
               wide array TargetRec {
                   longint;                     /* Record ID */
                   targetFileSpecFlags;         /* Target File Spec. Flags */
                   OSType;                      /* Target File Type */
                   OSType;                      /* Target File Creator */
                   unsigned hex integer;        /* Finder Flags */
                   unsigned hex longint;        /* Creation date of new file */
                   unsigned hex longint;        /* Modification date of new file */
                   rsrcID;                      /* Search Proc. Rsrc ID */
                   unsigned integer;            /* File Path Index in ist# resource */
               };
       };
};
```

Flag Descriptions

See flag descriptions of 'intf' resource for information about flags

Field Descriptions

File Paths Rsrc ID The ID of the 'ist#' resource containing the file paths for this 'itf#' resource. (2-bytes)

`Record ID` An ID that uniquely identifies this target file spec. When combining multiple 'intf' resource with the 'inf#' resource in the same Installer script, make sure no ID of an 'intf'

resource is used to identify a target file spec record in the 'itf#' resource. (4-bytes)

File Path Index — An index of the file path string within the specified 'ist#' resource. (2-bytes)

See field descriptions of 'infa' resource for information on the other fields

```
type 'ist#' {
       switch {
           case format0:
               key integer = 0;                 /* Data Format version */
               integer = $$Countof(StringArray);       /* Number of strings */
               wide array StringArray {
                   pstring;                     /* File Path */
               };
       };
};
```

Field Descriptions

File Path — The file path string for the target file spec record referencing this string index. This string must be a partial path starting with a colon or one of two reserved tokens (special-xxxx or folder-user). (pascal string, not even-padded)

Source File Spec Resource ('infs')

The 'infs' script resource is referenced from atoms.

```
#define fileSpecFlags                                                  \
       boolean    noSearchForFile, searchForFile;            \
       boolean    typeCrNeedNotMatch, typeCrMustMatch;       \
       fill bit[14]

type   'infs' {
       literal longint;                              /* File Type   */
       literal longint;                              /* File Creator   */
       unsigned hex longint;                       /* Creation Date*/
       fileSpecFlags;                                /* File Spec Flags */
       evenPaddedString;                             /* Full Path */
};
```

Flags descriptions

`noSearchForFile/searchForFile`
Ignored by Installer Engine.

`typeCrNeedNotMatch/typeCrMustMatch`
Determines if the type and creator contained in the File Creator and File Type fields must exactly match the source file on the source disk. Use the `typeCrMustMatch` flag to force the type and creator to match, otherwise the installation stops with an error.

Field descriptions

`Source File Type` — The source file's type. If the `TypeCrMustMatch` flag is used the source file's type must match the type entered in this field. Otherwise, the user is told the source disk is bad and the installation is canceled. (4-bytes)

`Source File Creator`	The source file's creator. If the `TypeCrMustMatch` flag is used source file's creator must match the creator entered in this field. Otherwise, the user is told the source disk is bad and the installation is canceled. (4-bytes)
`Creation Date`	The creation date the source file must have, otherwise the installation is stopped. Values of 0 or 1 are considered to match any creation date. (4-bytes)
`Source File Path`	The full path to the source file. (even-padded Pascal string)

About File Searching ('insp')

Whenever you want to update or delete a file on the user's disk but don't know where the file will reside, you'll need to search for it at installation time. For those File and Resource atoms you wish to search for the target file, attach a Search Procedure ('insp') script resource to the new Target File Spec. ('intf') script resource, then reference the 'intf' resource from the atom just as you would the older 'infs' script resource. Setting the `searchForFile` flag in the 'intf' resource will tell Installer Engine to call the code resource at the appropriate time. The built-in searching feature of Installer Engine will be removed and the `searchForFile` flag in the present 'infs' resource will be ignored.

The actual searching code will be supplied by you, the scriptwriter, in a code resource whose type and ID is specified in the 'insp' resource.

Using File Searching with File and Resource Atoms

The file searching code resource is given a pointer to a parameter block containing useful information about Installer Engine's environment. The search routine performs a search for the desired file(s), then passes back a list of those files it has found, including a result code to tell Installer Engine whether to continue or stop.

The search routine's task is to create and return a handle to an array of FoundFileRec records in the parameter block's `fFoundFiles` field. The array should contain zero or more FoundFileRec records that specify each target file to act upon. The number of elements in the array is determined by the size of the handle. If the array has no elements, then the atom is ignored. If one FoundFileRec element is specified, then the atom (file, resource, or font atom) is processed normally with the designated target file. If more than one FoundFileRec element is in the array, the atom is duplicated for each element.

The files that are returned to Installer Engine may or may not already exist. Installer Engine will verify that the volume and directory exist and are valid, and that the file name does not conflict with an existing directory name. If the file does not exist Installer Engine will create one, if necessary.

Using File Searching with Rule Clauses

File searching can also be used with Target File Specs that are referenced from rule clauses. If a Target File Spec returns more than one found file, the rule clause is duplicated for each valid file. The duplicated rule clauses are "anded" with the original rule clause. Therefore, in order for the original rule clause to return TRUE, every duplicated rule clause must also return TRUE.

Allowable Installer Functions During File Searching

While your file searching code resource is executing, you can call selected Installer functions.

Allowable Installer functions:

- Memory functions: INewHandle, IDisposHandle, ILockHandle, IUnLockHandle.
- Action Handler functions: RegisterAction.

File Searching Reference

This section describes the data structures and new script resources needed to support target file searching.

Data Structures

This section describes the function interface and parameter block passed to the Search Procedure code resource.

The Search Procedure must return a handle to an array of records describing the files it found. This array handle is defined as:

```
typedef struct {
   short   vRefNum;
   long        parID;
   Str63   name;
    short      fReferenceID;     /* Reserved for internal use. */
} FoundFileRec, FoundFileArray[], *FoundFileArrayPtr[], **FoundFileHdl[];
```

The handle should be allocated using Installer Engine's `INewHandle` routine.

Function Interface

This section describes the Search Procedure function interface and the required result values.

The file search function must return one of three results to tell Installer Engine how to proceed:

```
typedef enum { kFatalSearchError              = -1,
                  kSearchSuccessful              = 0,
                  kCancelSearchAndInstallation = 1
               } SearchResult;
```

The file search function must have the following interface:

```
SearchResult   FileSearchRoutine( SearchProcedurePBPtr);
```

Parameter Block

The file search code resource is passed a pointer to a parameter block containing useful information. Your code resource will want to return a handle to a list of found files in the fFoundFileArray field.

```
typedef  struct {
—>  ProcPtr           fCallBackProcPtr;
—>  short                fTargetVRefNum;
—>  long                 fTargetFolderDirID;
```

```
->  short                fSystemVRefNum;
->  long                 fSystemBlessedDirID;
->  long                 fRefCon;
->  OSType           fFileSpecType;
->  OSType           fFileSpecCreator;
->  long                 fFileSpecCrDate;
->  Str255;          fFileSpecPath;
<-  FoundFileArrayHdl fFoundFilesArray;
} SearchProcedurePBRec, *SearchProcedurePBPtr;
```

Field descriptions

`fCallBackProcPtr`	A pointer to Installer Engine's dispatch routine. You'll need to pass this field as a parameter to glue routines that provide access to Installer functions.
`fTargetVRefNum`	The target disk's vRefNum. When allowing the user to select a target application folder, this is the volume on which the folder resides.
`fTargetFolderDirID`	The target application folder's directory ID. This value is -1 if the user cannot select a target application folder. Target File Specs that use the reserved folder path `folder-user` will be placed in this folder.
`fSystemVRefNum`	The system disk's vRefNum. Target File Specs that use the reserved folder path `special-xxx` will be placed in the System Folder on this volume.
`fSystemBlessedDirID`	The directory ID of the System Folder on the disk with the refNum `fSystemVRefNum`. This directory is not necessarily the currently active System Folder.
`fRefCon`	A 4-byte value defined by the scriptwriter in the 'insp' script resource.
`fFileSpecType`	The value from the Target File Type field of the Target File Spec.
`fFileSpecCreator`	The value from the Target File Creator field of the Target File Spec.
`fFileSpecCrDate`	The value from the Target File Creation Date field of the Target File Spec.
`fFileSpecPath`	The value from the Target File Path field of the Target File Spec.
`fFoundFilesArray`	A handle to an array of found target files that is created and filled by the code resource.

Resource Description

The 'insp' script resource is referenced from the new Target File Spec. It defines how to call the code resource that will perform the actual search.

```
type   'insp' {
    switch {
        case format0:
            key integer = 0;  /* Search Procedure Format version */
```

```
            unsigned integer; /* Search Procedure Flags */
            literal longint;  /* Search Procedure Code Rsrc Type */
            literal integer;  /* Search Procedure Code Rsrc ID */
            literal longint;  /* RefCon Value */
            literal longint;  /* Required Free Memory */
            evenPaddedString; /* Search Procedure Summary */
};
};
```

Field descriptions

`Search Procedure Flags`	Currently reserved for use by Apple Computer, Inc. (2-bytes)
`Search Procedure Code Rsrc Type`	The type of the code resource that will be called to perform the search. (4-bytes)
`Search Procedure Code Rsrc ID`	The ID of the code resource that will be called to perform the search. (2-bytes)
`RefCon Value`	The value that will be passed to the code resource in the `fRefCon` field of the Search Procedure parameter block. (4-bytes)
`Required Free Memory`	The minimum number of free bytes the Search Procedure code resource needs in its own heap to make local allocation of memory using Macintosh OS/Toolbox routines. Enter 0 (zero) to not create a sub-heap and run inside Installer Engine's heap. Please see warnings about using sub-heaps in the Atom Extender section if you specify a value other than 0. (4-bytes)
`Search Procedure Summary`	An optional string briefly describing the purpose of this Search Procedure. This string is never displayed to the user. (even-padded Pascal string)

Using the Disk Order Resource ('indo')

By default, Installer Engine requests disks from the user using the order of the atoms specified in the script. This ordering is displayed in the status dialog. With more than just a few atoms, it can be very difficult to control the order of disks. To make this easier, if an Installer Disk Order resource exists in the script (resource type `'indo'`), this resource is used to determine the ordering of the disks. The format of the `'indo'` resource is given below. The resource contains a list of Pascal strings that are the names of the source disks. These names must exactly match the names of the volumes given in the source and target paths within the File Specs.

For example, let's say you have created a two disk set that uses one floppy disk and one CD volume. Your 'indo' might look like this:

```
resource 'indo' (1000) {
    format1 {{
        kExpectFloppyDisk,
        "Installer Disk",
        kExpectCDVolume,
        "Applications CD"
    }};
```

```
};
```

Specifying the expected type of source disk just helps Installer Engine know which type of disk drive to make available to insert the requested source disk. This feature does not affect the user's ability to create network installation folders on an AppleShare server volume.

Disk Order Reference

Resource Description

Format 0 of the new 'indo' resource:

```
#define SrcDiskType                                                    \
       integer                                                         \
       kExpectFloppyDisk     = 0,    /* It's a floppy Disk */ \
       kExpectCDVolume       = 1,    /* It's a CD */             \
       kExpectFoldersOnVol   = 2 /* It's a folder */

type 'indo' {
    switch {
        case format1:
            key integer = 1;
            integer = $$Countof(SrcVolArray);
            array SrcVolArray {
                   SrcDiskType;          /* Source Disk Type */
                   evenPaddedString;     /* Source Disk Name */
            };
    };
};
```

Field descriptions

`Source Disk Type`	The expected source disk type. Currently, a floppy disk; CD or folders on a CD, HD, or AppleShare volume. Installer Engine primarily uses this information to determine which type of disk to eject to make room for the requested source disk. For example, if the requested source disk is expected to be a CD and all CD drives are full, then one of the CDs will be ejected. (2-bytes)
`Source Disk Name`	The name of the source disk. The name can end in a colon if you wish. (2-bytes)

CHAPTER 5

Miscellaneous Resources

This chapter describes miscellaneous resources added to the scrip by, either the scriptwriter, or the ScriptCheck MPW tool.

Example (Part 3): Specifying Target and Source Files

Our example Installer script is almost complete. We need to add one additional resource, then run ScriptCheck.

Step 11: Add Installer Version Resource

Installer Engine version resource will make sure only Installer and Installer Engine applications open your Installer script that supports the features you are using. For example, if you've written your Installer script for Installer Engine 4.5.1, then use the following resource:

```
resource 'invs' (1) {
    format0 {
        0x4,
        0x51,
        release,
        0x0,
        "4.5.1"
    }
};
```

Step 12: Build the Installer Script File

If you've defined your resources using the Rez language, then you'll need to compile this text into an actual Installer script file. The "MakeFile" file inside the UserInterfaceExample folder in the Installer SDK provides an automatic way of building and ScriptChecking your Installer script.

Step 13: Run ScriptCheck on the Installer Script File

Before you begin using your Installer script file, you must run ScriptCheck on the file, if not already run in step 12. The ScriptCheck MPW tool verifies the integrity of your Installer script and updates various fields from the source files.

See the document *ScriptCheck 4.2 User's Guide* for more information about using this tool.

Step 14: Test the Installer Script

Now you're ready to test your Installer script. The "Single Installer Script Example" folder on the Installer SDK contains an Upgrader-based program that installs the UserInstallerExample Installer script. Feel free to use this Upgrader example as a starting point for your own Installer script.

About the Installer Version Resource ('invs')

You may want to ensure that the version of Installer Engine that you used for development and testing and that you shipped is the one that is launched. To do so, add an 'invs' Installer Version resource to specify the specific Installer version needed for your script. If you require Installer 4.0 and a different version is launched and the version specified in this resource is 4.0 then Installer Engine will display a dialog saying something like "The Installer document [your document's name here] requires version [invs version number here] of the Installer application. Try opening the Installer application that is in the same folder as this Installer document."

Installer Engine uses the four hex bytes that specify the version number to compare against its own version. The string is used only in the dialog that is displayed when the two versions do not match.

Installer Version Reference

Resource Description

Format 0 of the 'invs' resource:

```
type 'invs' {
    switch {
        case format0:
            key integer = 0;
            hex byte;         /* Major revision in BCD */
            hex byte;         /* Minor revision in BCD */
            hex byte;         /* Release Stage */
            hex byte;         /* Release Number */
            pstring;              /* Short Version Number  String */
    };
};
```

Field descriptions

`Major Revision`	The major version number. (2-bytes)
`Minor Revision`	The minor version number. (2-bytes)
`Release Stage`	The release stage. Four release stage constants are defined in the InstallerTypes.r file: development, alpha, beta, final and release. (2-bytes)
`Release Number`	The release number. (2-bytes)
`Short Version Number String`	Text description of version number. (even-padded Pascal string)

About the Script Size Resource (‘insz’)

Use of the new script resource, ‘insz’ with an ID of 1 is recommend for scripts over 10K in size. When the script resources are loaded into memory, they actually go into a sub-heap inside of Installer Engine’s main heap. This improves Memory Manager performance on older machines. In previous versions of Installer Engine, the size of this sub-heap was always overestimated to account for possibly compressed resources in the script file. This older scheme has the potential for wasting precious Installer heap memory, requiring a larger application partition.

ScriptCheck 4.X calculates this value and automatically places this resource into your script. The size calculated is calculated using the following script resource types: 'inaa', 'infa', 'inra', 'inff', 'inrm', 'intf', infs', 'inpk', 'inrl', 'infr', 'inex', 'invc', 'insp', 'inat', 'inbb', 'indo', incd', 'insz', 'icmt', 'inpc', 'inr#', 'itf#', 'ifa#', and 'ist#'.

Script Size Reference

Resource Description

Format 0 of the ‘invs’ resource:

```
type 'insz' {
       switch {
           case format0:
               key integer = 0;      /* Format version */
               unsigned longint;     /* Script Sub-Heap Size in bytes */
       };
};
```

Field descriptions

`Script Sub-Heap Size`	The size in bytes the script resources will require when loaded into memory. This value is used to create an optimally sized sub-heap for the Installer script resources. (4-bytes)

About the Script Creation Date Resource (‘incd’)

The script creation date resource holds a copy of the script file’s creation date field. This value is used by Installer Engine to correctly find and verify the source files when copied to an AppleShare server. Because the creation date of the source files may have been changed during the copy, Installer Engine uses the creation date in the ‘incd’ resource to calculate a delta value using the script’s creation date stored in the file by the File Manager. Without this resource, the user may get an error after beginning the installation saying that the source files could not be found.

ScriptCheck automatically updates or adds this resource to your script.

Script Creation Date Reference

Resource Description

The ‘incd’ resource:

```
type 'incd' {
       unsigned longint;     /* Script Creation Date in seconds. */
   };
};
```

Field descriptions

`Script Creation Date`	The original creation date in seconds of the script file. (4-bytes)

Installer Functions

Installer functions allow the code resource writer to communicate with Installer Engine and use utility routines provided by Installer Engine. Some of the routines are the essential mechanism for communicating with Installer Engine, such as when decompressing files during installation. These routines are often limited to being called from specific code resources. Other routines may be available all the time, either as a means to getting information from Installer Engine or as helpful utility routines.

Installer callback functions are actually glue routines that you must include in your code to access the proper routines inside Installer Engine. Either include the glue code file "InstallerCallbackGlue.c" in your source code or compile it separately and link to it. You'll find the declaration of all Installer functions in the interface file "InstallerScript.h".

Installer Functions Reference

Data I/O Routines

This section describes the routines available to manage reading and writing atom data. Most of these routines are only available in Atom Extenders. These routines function very similar to high-level File Manager routines. Consult *Inside Macintosh: File* s for an overview of topics such as positioning marks in files.

ReadSourceData

```
pascal OSErr ReadSourceData(    CallBackProcPtr   pCallBackProcPtr,
                            long*             count,
                            Ptr               bufferPtr );
```

`pCallBackProcPtr`	The callback pointer.
`count`	The number of bytes to read from the source data. After the call, the actual number of bytes read is returned in `count`.
`bufferPtr`	A pointer to a buffer of at least size `count`.

DESCRIPTION

The `ReadSourceData` function reads the specified number of bytes from the source data beginning at the current position. The supplied pointer must have already been allocated to at least the size of `count`.

RESULT CODES

`kNotImplementedErr`	30901	Routine not currently implemented
`noErr`	0	No error
`ioErr`	-36	I/O Error
`eofErr`	-39	End of part reached
`paramErr`	-50	Negative count or NULL bufferPtr

WriteTargetData

```
pascal OSErr WriteTargetData(    CallBackProcPtr   pCallBackProcPtr,
                                 long              count,
                                 Ptr               bufferPtr );
```

pCallBackProcPtr	The callback pointer.
`count`	The number of bytes to write to the target at the current position.
`bufferPtr`	A pointer to a buffer of at least size `count`.

DESCRIPTION

The `WriteTargetData` function writes the specified number of bytes from the supplied buffer to the target beginning at the current position. The data is not immediately written to the target file, but rather is buffered using Installer Engine heap or MultiFinder temporary memory. This reduces our need for the target disk, which is especially important when installing onto floppy disks.

IMPORTANT

Since there is overhead associated with each call to `WriteTargetData`, calls to write very small amounts of data (less than 1K) should be avoided.

RESULT CODES

`kNotImplementedErr`	30901	Routine not currently implemented
`noErr`	0	No error
`paramErr`	-50	Negative count or NULL bufferPtr
`memFullErr`	-108	Not enough memory to buffer data

ReadTargetData

```
pascal OSErr ReadTargetData(     CallBackProcPtr   pCallBackProcPtr,
                                 long*             count,
                                 Ptr               bufferPtr );
```

pCallBackProcPtr	The callback pointer.
`count`	The number of bytes to read from the target data. After the call, the actual number of bytes read is returned in `count`.

bufferPtr	A pointer to a buffer of at least size count.

DESCRIPTION

The ReadTargetData function reads the specified number of bytes from the target data beginning at the current position. The supplied pointer must have already been allocated to at least the size of count.

IMPORTANT

The ReadTargetData function can only be called while in a Version Compare code resource that was referenced from a Resource Atom. Calling this function at any other time will return a kNotImplementedErr error.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error
ioErr	-36	I/O Error
eofErr	-39	End of part reached
paramErr	-50	Negative count or NULL bufferPtr

SetTargetDataPos

```
pascal OSErr SetTargetDataPos(   CallBackProcPtr   pCallBackProcPtr,
                                 short             positionMode,
                                 long                  positionOffset );
```

pCallBackProcPtr	The callback pointer.
positionMode	The positioning mode.
positionOffset	The positioning offset.

DESCRIPTION

The SetTargetDataPos function sets the current position in the target data.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error
eofErr	-50	End of target data reached
posErr	-40	Attempt to position mark before start of target data

GetTargetDataPos

```
pascal OSErr GetTargetDataPos(   CallBackProcPtr   pCallBackProcPtr,
                                 long*                 positionOffset );
```

pCallBackProcPtr	The callback pointer.
positionOffset	On output, contains the current position offset from the beginning of the target data.

DESCRIPTION

The GetTargetDataPos function returns the current position from the beginning of the target data.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error

GetTargetDataEOF

```
pascal OSErr GetTargetDataEOF(   CallBackProcPtr   pCallBackProcPtr,
                                 long*             theLength );
```

pCallBackProcPtr — The callback pointer.

theLength — On output, contains the current length of the target data.

DESCRIPTION

The `GetTargetDataPos` function returns the current length of the target data.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error

SetSourceDataPos

```
pascal OSErr SetSourceDataPos(   CallBackProcPtr   pCallBackProcPtr,
                                 short             positionMode,
                                 long              positionOffset );
```

pCallBackProcPtr — The callback pointer.

positionMode — The positioning mode.

positionOffset — The positioning offset.

DESCRIPTION

The `SetSourceDataPos` function sets the current position in the source data.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error
eofErr	-50	End of source part data reached
posErr	-40	Attempt to position mark before start of source part data

GetSourceDataPos

```
pascal OSErr GetSourceDataPos(   CallBackProcPtr   pCallBackProcPtr,
                                 long*             positionOffset );
```

pCallBackProcPtr — The callback pointer.

positionOffset — On output, contains the current position from the beginning of the source data.

DESCRIPTION

The `GetSourceDataPos` function returns the current position from the beginning of the source data.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error

GetSourceDataEOF

```
pascal OSErr GetSourceDataEOF(   CallBackProcPtr   pCallBackProcPtr,
                                 long*             theLength );
```

pCallBackProcPtr — The callback pointer.

theLength — On output, contains the length of the source data.

DESCRIPTION

The GetSourceDataPos function returns the length of the source data.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error

Memory Routines

This section describes the routines available to manage static memory. Some code resources can pass back a handle to static memory so the code resource can communicate between invocations of the code resource. Generally, these routines are available all the time.

INewHandle

```
pascal Handle INewHandle(    CallBackProcPtr   pCallBackProcPtr,
                             long              newHandleSize );
```

pCallBackProcPtr — The callback pointer.

newHandleSize — The requested size to allocate the handle.

DESCRIPTION

The INewHandle function attempts to allocate a handle of the specified size. If the memory is not immediately available, Installer Engine writes out any buffered target data to the appropriate target disk, then tries again. The resulting handle is NULL if the handle could not be allocated.

Use INewHandle when you want to pass information between invocations of an Atom Extender using the fStaticDataHandle of the parameter block.

IMPORTANT

Always use IDisposHandle to release a handle allocated using INewHandle. Always use IHLock and IHUnLock when locking and unlocking a handle allocated using INewHandle.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error
memFullErr	-108	Not enough room in Installer Engine's or MultiFinder's heap zone

IDisposHandle

```
pascal Handle IDisposHandle(     CallBackProcPtr   pCallBackProcPtr,
                                 Handle            storageHandle );
```

pCallBackProcPtr	The callback pointer.
storageHandle	The handle to be disposed.

DESCRIPTION

The `IDisposHandle` function disposes the handle in the parameter `storageHandle`. The handle could have been allocated using Installer Engine's `INewHandle` function, or with any other Macintosh Toolbox routine.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error
nilHandleErr	-109	NULL master pointer
memWZErr	-111	Attempt to operate on a free block

IHLock

```
pascal void    IHLock( CallBackProcPtr  pCallBackProcPtr,
                   Handle storageHandle );
```

pCallBackProcPtr	The callback pointer.
storageHandle	The handle to lock.

DESCRIPTION

The `IHLock` function locks the specified handle.

IMPORTANT

Always use `IHLock` to lock a handle allocated using `INewHandle`.

RESULT CODES

kNotImplementedErr	30901	Routine not currently implemented
noErr	0	No error
nilHandleErr	-109	NULL master pointer
memWZErr	-111	Attempt to operate on a free block

IHUnLock

```
pascal void    IHUnLock(  CallBackProcPtr   pCallBackProcPtr,
                      Handle            storageHandle );
```

pCallBackProcPtr	The callback pointer.
storageHandle	The handle to lock.

DESCRIPTION

The `IHUnLock` function unlocks the specified handle.

IMPORTANT

Always use `IHUnLock` to unlock a handle allocated using `INewHandle`.

RESULT CODES

`kNotImplementedErr`	30901	Routine not currently implemented
`noErr`	0	No error
`nilHandleErr`	-109	NULL master pointer
`memWZErr`	-111	Attempt to operate on a free block

Misc. Routines

This section describes various miscellaneous routines available.

MakeFSSpecFromFileSpecID

```
pascal OSErr MakeFSSpecFromFileSpecID ( CallBackProcPtr   pCallBackProcPtr,
                                        short pFileSpecID,
                                        FSSpec *pFSSpecPtr,
                                        StringHandle *pExtraPathInfo)
```

pCallBackProcPtr — The callback pointer.

DESCRIPTION

The `RegisterScriptAction` function.

IncrementStatusBar

```
pascal void IncrementStatusBar( CallBackProcPtr   pCallBackProcPtr,
                                short                 pIncrementAmount )
```

pCallBackProcPtr — The callback pointer.

pIncrementAmount — Number of increments.

DESCRIPTION

Calling the `IncrementStatusBar` function increments the status bar during long waits inside your code resource. Most often you'll use this in your Action Atom code resource. Upon entry to your Action Atom you are given 100 status increments to use to show progress while executing a single Action Atom. Installer Engine limits the bar so you don't have to worry about incrementing the bar too far.

InstallerFindSpecialFolder

```
pascal OSErr InstallerFindSpecialFolder(
                        CallBackProcPtr   pCallBackProcPtr,
                        short                pVRefNum
                        OSType            pFolderType
                        Boolean           pCreateFolder
                        short   *            pFoundVRefNum
                        long    *            pFoundDirID );
```

pCallBackProcPtr — The callback pointer.

pVRefNum — The volume reference number

pFolderType	The folder type being looked for
pCreateFolder	true if the folder should be created if not found
pFoundVRefNum	The volume reference number returned
pFoundDirID	The directory ID returned by FindFolder

DESCRIPTION

The InstallerFindSpecialFolder function attempts to locate or create the folder which type is pFolderType in the fld# or nfd# resource. The fld# resource is used for System 7.x installations (target system), the nfd# resource is used for System 8.x installations. If your installer script contains references to System 8 folder types and you are installing on top of System 7, Installer Engine will use a resource of type 'nf7#' (same format as nfd#) to locate the folder.

RESULT CODES

fnfErr	-43	Type not found
	-1	Error looking for fld# or nfd#
noErr	0	No error

GetBoxFlagOverwrite

```
pascal long GetBoxFlagOverwrite(InstallerCallBackUPP pCallBackProcPtr);
```

pCallBackProcPtr	The callback pointer.

DESCRIPTION

The GetBoxFlagOverwrite function returns the value of the box flag overwrite passed to Installer Engine by the Upgrader application (or the Script Editor). This allows script writers to take appropriate actions in their action atoms, relying on a machine ID passed by the Upgrader, and not the real machine ID as returned by Gestalt.

If no box flag was passed by the client application, this function will return the real machine ID as returned by Gestalt.

IsLaunchedByUpgrader

```
pascal Boolean IsLaunchedByUpgrader(InstallerCallBackUPP pCallBackProcPtr);
```

pCallBackProcPtr	The callback pointer.

DESCRIPTION

Currently, Installer Engine's IsLaunchedByUpgrader function always returns TRUE regardless of the client application that launched it.

IsParasiteScript

```
pascal Boolean IsParasiteScript(InstallerCallBackUPP pCallBackProcPtr);
```

pCallBackProcPtr The callback pointer.

DESCRIPTION

The IsParasiteScript function returns TRUE if the current script is a parasite script. This call only makes sense if Installer Engine was launched by the Upgrader 1.1.1 or later application.

NOTE : The return value is only significant if Installer Engine is used in conjunction with Upgrader 1.1.1 or later.

PresentMessageAlert

```
pascal SInt16 PresentMessageAlert (InstallerCallBackUPP pCallBackProcPtr,
                                   SInt16 pDialogID,
                                   ConstStr255Param pMessageStr,
                                   SInt16 pCancelButtonID,
                                   SInt16 pDefaultButtonID );
```

pCallBackProcPtr The callback pointer.

DESCRIPTION

When it's absolutely necessary to communicate with the user from within your Installer script, Installer Engine provides a callback routine to request that the client show a dialog. Since it's only a request, you are at the mercy of the client to display your dialog. The PresentMessageAlert callback routine will return -1 if the client could not display the dialog; otherwise, the clicked button number is returned.
The following code example uses the PresentMessageAlert callback routine.

```
SInt32 MyActionAtom( … )
{
    SInt16     theErr;
    SInt16     theButtonClicked;

    theButtonClicked = PresentMessageAlert( callBackPtr, 200, "\pHello!", 2, 1 );

    if( theButtonClicked != -1 )
    {
        // Interpret the user's response

    }
    else
    {
        // Do default action here
        switch( theButtonClicked )
        {
            case 1:    …
        }

        // Don't return an error if the default action succeeded
        theErr = noErr;
    }

    return (SInt32)theErr;
```

```
}
```

RegisterScriptAction

```
pascal void RegisterScriptAction( CallBackProcPtr  pCallBackProcPtr,
                                  short                actionClassID,
                                  short                actionIdentifier,
                                  void*                param0,
                                  void*                param1,
                                  void*                param2,
                                  void*                param3,
                                  void*                resultPtr )
```

pCallBackProcPtr	The callback pointer.
actionClassID	The action class ID number.
actionIdentifier	The action identifier number.
param0...param3	A pointer to data or 4-byte value used to send information to the Action Handler about the action.
resultPtr	A pointer to a variable that the Action Handler can send information back to the caller. The format of the data structure is defined individually for those actions that allow modification to the result.

DESCRIPTION

The `RegisterScriptAction` function registers an action with Installer Engine, which will be immediately given to all loaded Action Handlers. This function is made available to help code resource writers easily send text messages to Installer Debugger, using the action class kDebuggingAction, and the action identifier kGenericDebugActID.

CHAPTER 7

Runtime Issues

This chapter discusses information that may help some scriptwriters keep out of trouble and provide solutions to unique situations. This chapter also provides a listing of the built-in error strings that can be return during and after an installation or removal.

Installer Script Compatibility

The majority of existing Installer scripts (version 3.3 and newer) should run unmodified with Installer Engine. The most common compatibility problem is a code resource attempting to communicate directly with the user by displaying dialog during the installations.

Interacting with the User

Installer scripts that display dialogs using toolbox routines from within code resources, such as action atoms, will not work when running with Installer Engine. For existing Installer scripts that call select toolbox routines to create or show a window, Installer Engine attempts to trick the code resource into not displaying the dialog, and allows the installation to complete. Writers of new Installer scripts can use a callback routine to interact with the user, although such user interaction is discouraged.

Compatibility for Existing Installer Scripts

Installer Engine prevents the scriptwriter from displaying a window by patching a handful of window creation routines so that they return a value that can be checked by your code. The following routines are patched to return the specified result.

Patched Routine	Result under Installer Engine
GetNewDialog	Returns NULL
NewDialog	Returns NULL
NewWindow	Returns NULL
Alert	Returns -1
StopAlert	Returns -1
NoteAlert	Returns -1

CautionAlert	Returns -1

Properly written action atoms normally check that the dialog or window pointer is not NULL before continuing. When a NULL pointer is discovered the programmer normally returns a fatal error result from the action atom, preventing the installation from completing successfully. To combat this problem, Installer Engine flags whether an action atom attempts to show a dialog, and if a fatal error is returned ignores this error allowing the installation to continue.

The following code example shows how a scriptwriter might write any action atom code resource that displays a dialog when running with Installer 4.0.X, but does not display the dialog when running with Installer Engine.

```
SInt32 MyActionAtom( … )
{
    SInt16     theErr;
    DialogPtr  theDialog;

    theDialog = GetNewDialog( 200, NULL, (WindowPtr)-1L );

    if( theDialog != NULL )
    {
        // Display dialog
        theErr = HandleMyDialog( theDialog );

    }
    else
    {
        // Do default action here

        // Don't return an error if the default action succeeded
        theErr = noErr;
    }

    return (SInt32)theErr;
}
```

Message Area Strings

The following error messages may be returned by Installer Engine.

1. **Message**: An unknown error #^0 has occurred.

   Installer Engine doesn't have a specific explaination for the problem.

2. **Message**: A problem occurred with the file server. You may not have the necessary privileges. Check with your network administrator.

   Any one of several AFP errors was returned by the System.

3. **Message:** A needed file is already in use or was left open.

   We may display this error if we get a -47 or -49 File Manager error back from our Target File routines. This error may be converted into a more generic error.

4. **Message:** You do not have the necessary privileges to access all of the files needed on the server. Contact your network administrator.

   The user is installing from a Network folder and does not have the necessary access privileges to all of the required folders. As specified in the alert message, the user needs to contact the AppleShare administrator to set folder privileges appropriately.

5. **Message:** A disk error has occurred and installation cannot continue. Your disk may be damaged.

   This error occurs when the Installer and/or script file are placed on an MFS formatted diskette, and are used from that diskette. It is also mapped to File Manager error number –123 through –120 in some situations.

6. **Message:** An AppleShare volume has unexpectedly disappeared. The installation cannot continue.

   The user was installing from the network and the network connection became broken. The user needs to contact the network administrator to have the connection problem diagnosed.

7. **Message:** The Installer needs more memory to perform an installation. Try quitting other applications and running the Installer again. You could also try removing some system extensions and restarting to make more memory available.

   We display this error when the Memory Manager returns an error from -117 to -108. Depending on where we run out of memory, we'll display this error if the Installer's partition is too low.

8. **Message:** A problem occurred with the AppleTalk network. Please quit the Installer restart your computer, and try running the Installer again.

   The network connection may have gone down while performing a network installation.

9. **Message:** A necessary file is locked. Installation cannot continue.

   The Installer received the File Manager error -54 or -45 while opening a target file. Most likely, something or someone locked a target file during the installation.

10. **Message:** One of your disks is no longer available. Please quit the Installer, restart your computer, and try running the Installer again.

    We may display this error when the File Manager returns an error of -53. This error may be converted into a more generic error.

11. **Message:** The disk is locked.

    The alert occurs only on an installation to a 'Floppy only' CPU. Between the switching of the source and target disks, the user has locked the target disk.

12. **Message:** There are too many files open on your computer. Please quit any other applications you are running and try again.

    The Installer received the File Manager error -42 while opening a file. This might be the case if another application has numerous files open, which would cause the Installer to reach the maximum number of files that can be open at once. Try using up all but two FCBs then performing a non-live installation.

13. **Message:** The destination disk "^3" is too full. Please remove some files you no longer need, and try the installation again.

    This problem can occur when the target volume is being shared via File Sharing. If an installation is started and a user logs onto the same disk and copies additional data to the server after the Installer preflight has taken place, the target disk may run out of hard disk space resulting in this error.

14. **Message (2802):** Installer Engine cannot open the document "^0" because it is the wrong type.

    The document you are trying to open is not one of the Installer script file types that Installer Engine can open.

15. **Message (2242):** The files needed for this installation could not be found. The Installer is looking for a disk or folder named "^0". Please find the correct folder or disk and try again.

    Generic, can't find the source disks/folders problem.

16. **Message (2302):** Installer Engine cannot run on this model of computer. Please contact your authorized Apple dealer and inquire about upgrading your hardware.

    This error occurs when trying to use Installer Engine on a 128K or 512K Mac. Installer Engine is not compatible with these Systems.

17. **Message (2501):** The feature "^0" could not be found.

    Couldn't find feature by name when using Apple events to retreive information about a feature set's features.

18. **Message (2502):** Recommended installation not supported by Installer document.

    This Installer script does not contain an easy feature set or 'infr' resource of ID 764.

19. **Message (2302):** Installer Engine could not find the ObjectSupportLib library file.

    Installer Engine requires the ObjectSupportLib library to run. If your Installer runs on systems prior to Mac OS 8.0, then you must place the ObjectSupportLib library file suppliced on the Installer SDK in the same folder as the Installer Engine.

20. **Message (2204):** There is not enough space on the disk "^3" to complete the installation (^0K needed, ^1K available). Remove some items from the disk "^3" and try again. Instead, you could try installing on another disk.

    This message occurs when there is insufficient disk space to perform the installation. In this case, the user is not installing into the current active System Folder.

21. **Message (2244):** The file "^0" on the disk "^3" is too large to accommodate the installation. The installation continue.\n\nEither, start your computer with Mac OS 7.6 or later then try installing again, or perform a clean installation.

    We can't add any more resources to a file because it has reached the limit. Booting under a different verison of SSW works because newer versions allow more resources to be stored in the file solves this.

22. **Message (2027):** The Installer needs to create a file named "^0" on the disk "^3" but a folder with this name already exists.\n\nPlease rename the folder or move it so it is inside another folder.

    The Installer is unable to a file because a folder already exists with the same name in the specified location.

23. **Message (2211):** There was nothing to remove.

    This message occurs when the user goes to perform a software removal on a volume that does not have the designated software to remove. Action Atoms that fire on removal will prevent this message from appearing and will present the standard "Removal was successful" message.

24. **Message (2217):** There were files missing on "^3" that are required for this installation. Please check your manual to see what is required.

    The Installer script is trying to install resources or fonts into several files, but the files do not exist and must because the "tgtRequired" flag is set in the atoms.

25. **Message (2218):** The file "^0" is missing on "^3" but is required for this installation. Please check your manual to see how to get that file.

    The Installer script is trying to install resources or fonts into the specified file, but the file does not exist and must because the "tgtRequired" flag is set in the atom.

26. **Message (2219):** A file is missing from "^3" but is required for this installation. Please check your manual to see what files are required and how to get these files.

    The Installer script is trying to install resources or fonts into a file, but the file does not exist and must because the "tgtRequired" flag is set in the atom. The name of the file is unknown or is zero length.

27. **Message (2220):** System Software is required for this installation but is not present on "^3". OK, then install System Software before or as part of this installation.

    The Installer script is trying to install a resource or font into a "System" file, but the file does not exist. The file is required as the "tgtRequired" flag is set in the atom.

28. **Message (2229):** Cannot overwrite a protected resource in the file "^0" on the disk "^3".

    A resource specified for replacement on the target disk has the protected bit set. The protected bit needs to be cleared before the installation can proceed.

29. **Message (2235):** No installation was necessary.

    After clicking the Install button it has been determined that nothing needs to be installed. This might happen if the script is only updating one file and the "keepExisting" flag in the File Atom is being used.

30. **Message (2236):** "<DiskName>" has the correct name but is not the correct disk. Click OK, then try inserting a different disk with this name..

    The Installer was unable to find files that the designated disk was supposed to contain. Some possible causes: 1. A file name on the disk has been modified by having a space character appended to the end of the file name. 2. The contents of the disk are different than those specified by the Installer script. This second problem can occur if a user tries to update an Installer disk set by replacing an existing file with a more recent version. When the Installer has the creation date/time setting stored for a file in the script, and that time stamp does not match that of the source file, the Installer reports this error. 3. A final cause is that a user has tried to set up a network installation on a Novell server running Netware 2.x or earlier. See Appendix A for a discussion of this problem.

31. **Message (2800):** The Installer document "^0" is damaged. Make sure you are using the original Installer disks and try again.

    The script document cannot be opened by the. This error can result given a network installation when the administrator launches the script locally, then a user accesses the script from a workstation and tries to run the script using a remote copy of the Installer. The workstation Installer will receive this alert message.

    Another possible reason is that not enough memory is available to load the script into the Installer's heap. Try raising the Installer's partition if the script is large.

32. **Message (2232):** Problems were encountered reading the source file "^0". Installation cannot continue.

    Generic source error.

33. **Message (2227):** Problems were encountered accessing the file "^0" on the disk "^3". Please move the file to another folder and try again.

    Generic target error.

34. **Message (2231):** There is a problem with a disk you are installing onto. No installation can take place. Try installing onto another disk.

    Resource Manager errors were returned during opening, reading or writing a target resource or font. Most likely, the target file's resource fork is corrupted.

35. **Message (3209):** To ensure safe removal from the active startup disk "^3" ^1K of free disk space is required (^0K available).\n\nPlease remove some files using the Finder and try again.

    This message occurs when there is insufficient disk space to remove files from the active System Folder. This sounds weird, but the Installer can not delete resources from active files, or files that are currently active. (51)

36. **Message (3204):** There is not enough space on the active startup disk "<DiskName>"to do this installation (<#> K available). To install on this disk you will need ^1K free.

    This message occurs when there is insufficient disk space to perform the installation. When installing into the active System Folder, the Installer takes additional steps to guarantee disk integrity should the installation be canceled or aborted for whatever reason.

    For example, to install a 100 byte 'adbs' resource to the System file, the Installer moves the System file to the temporary folder in the System folder, then makes a copy back in the original location. The installation occurs on the copied (and currently non-opened) System file. If a problem occurs, or the users cancels the installation, the copied file is deleted, and the System file in the temporary folder restored. If the installation is successful, the Installer places a Cleanup INIT which deletes the temporary folder and it's contents at startup. Adding a 100 byte resource, may require 2M or greater of hard disk memory on the boot volume. For this reason, it is difficult to perform an installation of some System resource to a boot floppy. If the target volume is not a boot volume, these precautions are not performed.

37. **Message (2238):** An error occurred while trying to complete the installation.

    A action returned a fatal error.

38. **Message (2243):** The installation was stopped by the Installer script.

    The action atom was cancelled, either by the user and the action atom didn't present it's own dialog, or the action atom itself.

39. **Message (2028):** The Installer needs to create a folder named "^0" on the disk "^3" but a file with this name already exists.\n\nPlease rename this file or move it to another folder.

    A file with the same name as a folder must be created during the installation. For example, place file named "Control Panels" into the System Folder, then try to install one or more Control Panels into the System Folder.

40. **Message (2240):** The Installer needs to modify the file named “^0” on the disk “^3” but that file is locked. Please unlock the file and try again.

    Depending on the scriptwriter’s wishes, the Installer will not overwrite locked files. Unfortunately, if several files are locked, this alert occurs for each locked file.

41. **Message (2803):** Installer Engine cannot open an alias to an Installer document.

    Installer Engine does not support opening aliases to Installer scripts.

42. **Message (2301):** Installer Engine does not support this computer model or the version of system software.

    Installer Engine does not support the Macintosh model or the version of SSW. Installer Engine requires a 68020 processor and System 7.0.

43. **Message (2804):** Installer documents cannot be printed.

    The user has tried to print the document using the Finder or via the print Apple event.

44. **Message (2223):** Problems were encountered deleting old files. Please quit the Installer, restart your computer and try running the Installer again.

    This problem occurs when the Installer is unable to modify a target document. On a non-boot volume installation, an open application may have some file opened that is targeted for replacement by the installation. For example use ResEdit to open the System file on a non-boot volume, and try to install system software to that volume.

45. **Message (2241):** The files needed for this installation could not be found on your server volume "<VolName>". Please contact your network administrator about this problem.

    This error occurs when a network installation is run, and an error occurs accessing the source files. For example move some of the necessary files from the network installation folder, and attempt the installation.

46. **Message (2805):** The Installer document “^0” cannot be opened because it requires version ^1 of the Installer Engine application.

    The installation script includes an 'invs' resource which specifies the exact version of the Installer to use. The version of the Installer opened does not match the script's 'invs' resource. One cause of this alert is that there exists another version of the Installer in the Finder's application database, and the user launched the script instead of the Installer application.

47. **Message (2215):** The file “^0” on the disk “^3” is too large to accommodate the installation. The installation continue.\n\nEither, start your computer with Mac OS 7.6 or later then try installing again, or perform a clean installation.

    The script is adding resources or fonts to a file whose resource fork will exceed the 16Mb size limit. This was a real issue when installing Kanji fonts into the System file, but shouldn’t happen as often with separate font suitcases.

48. **Message (2029):** The installation cannot continue because the source file “^0” on the disk “^1” will be overwritten during the installation. Please choose a different destination disk and try again.

    The Installer will write over a file that is part of the installation sources. For disk mode only.

49. **Message (2030):** The installation cannot continue because the source file "^0" on the disk "^1" will be overwritten during the installation. Please select a different destination folder and try again.

    The Installer will write over a file that is part of the installation sources. For folder mode only.

50. **Message (2031):** The removal cannot continue because the source file "^0" on the disk "^1" will be deleted during the removal. Please choose a different destination disk and try again.

    The Installer will delete a file that is part of the installation sources. For disk mode only.

51. **Message (2032):** The removal cannot continue because the source file "^0" on the disk "^1" will be deleted during the removal. Please select a different destination folder and try again.

    The Installer will delete a file that is part of the installation sources. For folder mode only.

52. **Message (3412):** The Installer cannot create files on the disk named "^0" because you do not have write access. Please choose a different destination disk.

    The user doesn't have write permission to the root level of the target volume. This might be the case when installing onto an AppleShare volume or UNIX volume.

53. **Message (3410):** No destination disk has been chosen.

    There's no valid disk to install onto.

54. **Message (3413):** The selected disk named "^0" is locked. Please choose a different destination disk.

    A locked hard drive or perhaps some third-party locked medium has been selected. To replicate, use Apple HD SC Setup to lock a volume and try to perform an installation on it. A CD-ROM disk or perhaps a locked Syquest disk will also produce this message.

55. **Message (3414):** The selected disk named "^0" is locked. Either, unlock the disk, or choose a different destination disk.

    The Installer cannot write to a locked destination disk.

56. **Message (3411):** The selected disk named "^0" is not an HFS volume. The Installer cannot install on volumes without the hierarchical file system. Please choose different destination disk.

    The Installer cannot install to an unlocked MFS 400K floppy disk.

57. **Message (3415):** The selected disk named "^0" is a network server volume. The Installer cannot install on network server volumes. Please choose a different destination disk.

    The Installer cannot install software to a mounted AppleShare volume unless the preference flag has been set to allow this.

58. **Message (3409):** The selected disk named "^0" cannot be a destination disk. Please choose a different destination disk.

    An unknown error prevents installation onto target (App Folder mode).

59. **Message (3405):** The Installer cannot create files on the disk named "^0" because you do not have write access. Please choose a different destination folder.

    The user doesn't have write permission to the root level of the target volume. This might be the case when installing onto an AppleShare volume or UNIX volume (Folder Mode).

60. **Message (3407):** The disk named "^0" is locked. Please choose a different destination folder.

    The Installer cannot write to a locked target disk (App Folder Mode).

61. **Message (3408):** The disk named "^0" is a network server volume. The Installer cannot install on network server volumes. Please choose a different destination folder.

    The Installer cannot install software to a mounted AppleShare volume unless the preference flag has been set to allow this (App Folder Mode).

Installer Apple Event Suite

The Apple event suite implemented in Installer Engine has been designed to accommodate a variety of client application needs. In addition to supporting AppleScript, custom client applications can be created to provide an appropriate user experience to accommodate a particular installation solution. An example of such an application is the Upgrader application Apple uses to install Mac OS 8.5.

This chapter provides a summary of the Installer Apple event suite and touches on issues that client applications should take in to account. Two AppleScript examples are provided at the end of this chapter for those the need a simple automation solution.

Apple Event Suite Summary

Communicating with Installer Engine requires a combination of Apple events sent and received, as well as manipulation of Apple event objects.

Sending Apple Events to Installer Engine

Clients will use seven Apple events handled by the application to perform basic tasks, such as starting and stopping and installations. Three additional Apple events are used to get data, set data, and count information contained in objects.

Apple Event	Description
Install [feature IDs {}] [feature names {}] [preinstallation report] [preinstall report only]	Start the installation.
Remove [feature IDs {}] [feature names {}] [pre removal report] [pre removal report only]	Start the removal.
Cancel	Cancel the installation that is currently in progress.
Register Client [with basic progress] [with progress] [with debugging] [with reporting]	Adds a process to the list of clients receiving the specified class of events.
Deregister Client	Deletes a process from the list of clients receiving events.
Open	Opens an Installer script document.
Quit	Quits Installer Engine. If an installation is currently in progress it will wait until it has completed before quitting.

Set Data	Sets the value of a property of an object.
Get Data	Retrieves the value of a property and object.
Count	Retrieves the number elements of an object.

Starting an Installation or Removal

An install event without parameters requests that the recommended installation be started. Installation of specific features can be initiated by providing a list of the specific features to be installed as a parameter of the install event.

A removal can be initiated with the remove Apple event. A list of features of remove should always be provided with the remove event.

Canceling an Installation

An installation currently in progress can be canceled at any time using the cancel Apple event.

Registering and Deregistering Clients

Clients that wish to receive specific types of progress or debugging events must register themselves with Installer Engine using the register client Apple event. Such clients may include: debugging applications, remote clients, or any client that wants to monitor Installer Engine.

Use the deregister client Apple event to stop Installer Engine from sending further progress and debugging events to the client.

Opening Installer Script Documents

A client can open an Installer script document by sending an open Apple event with a reference to the Installer script file. If a document is already open, it is automatically closed before opening the specified Installer script.

Quitting Installer Engine

The Installer Engine application can be quit by sending a quit Apple event to it.

How Installer Engine Processes Events

Users familiar with scripting the Finder (and most other scriptable applications), know that most events are synchronous — they do not return control to the client application until the action is finished, or failed trying. This makes it easier to write AppleScripts to perform numerous tasks one after another, but makes it harder for sophisticated clients to manage long tasks.

The duration of an installation typically depends on how many objects are being installed, and the speed of the source and target media. For this reason, we have chosen to make the Install and Remove events asynchronous — they return immediately, even though the installation may take many minutes. Because the client regains instantaneous control after starting the installation, it can monitor the installation process as if feels necessary without having to use an idle routine or worrying about event time-out issues. Some clients will find that querying the

cStatus object provides enough feedback, while other clients may want detailed progress information sent to them. These topics are explored in the section titled “Advanced Interaction.”

While Installer Engine is performing an installation certain actions cannot be performed. These restricted events generally change the environment, which must stay constant until the installation finishes. If you try to send an Apple event that is restricted it will return immediately with an error. Certain events, such as the cancel Apple event, are necessary to allow the client to control the installation.

Events that cannot be sent during the installation or removal process:

- Install — starting another installation must wait for the current one to finish.
- Remove — starting another removal must wait for the current one to finish.
- Open — opening another document cannot happen until the current one is able to be closed.
- Setting any cDocument properties — the document properties provide the environment settings for the installation, and therefore cannot be modified.

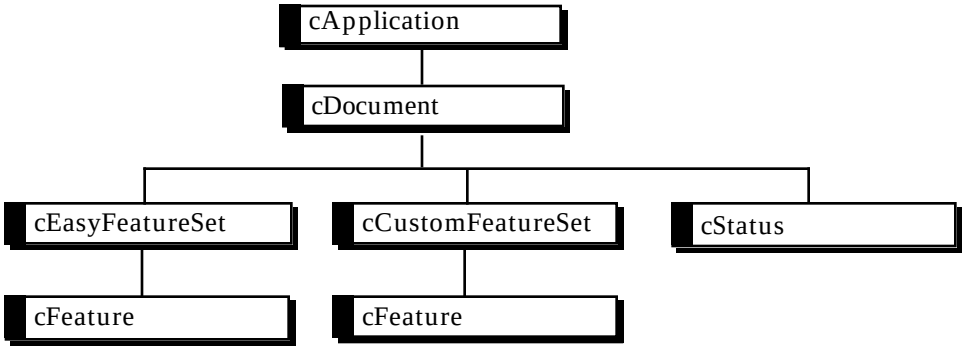
Events allowed during the installation or removal process:

- Cancel — an installation can be canceled at any time.
- Quit — the installation or removal is completes, then the application is quit.
- Register Client and Deregister Client — the client can change what events are being sent to it by Installer Engine during an installation or removal.
- Getting any property value — all object property values can be read during the installation.

Installer Engine Objects

Installer Engine uses a hierarchy of Apple event objects, see Figure 1-1, to describe the contents of the Installer script and manage environmental information, such as the selected destination disk.

Figure 8-1 Apple event object hierarchy for Installer Engine



Application Object

The application object provides access to the currently open Installer script and handles the following events: Install, Remove, Cancel, Register Client, Deregister Client, Open and Quit.

Property	Data Type	Mod.	Description
pVersion	typeVersion	R/O	The version number of Installer Engine.
pName	typeChar	R/O	The name of Installer Engine.

Element	Access Form	Description
cDocument	formAbsolutePosition	The currently open Installer script document.

Document Object

When an Installer script is open, its document object is accessible through the application object. Only one document can be open at any one time.

Property	Data Type	Mod.	Description
pName	typeChar	R/O	The name of the Installer Script.
pAppTarget	cFile	R/W	The target application folder. If the script uses disk user mode, then only volume is used.
pSysTarget	cFile	R/W	The target volume which contains the System Folder to be installed into. The parent director and folder name are ignored.
pReplaceNewerFiles	typeBoolean	R/W	If TRUE, allows newer files to be replaced with older files without an error being generated.
pReplaceLockedFiles	typeBoolean	R/W	If TRUE, the Installer allows locked files to be replaced without an error being generated.
pSourceLocation	cFile	R/W	The folder that contains the source folders.
pDoCleanInstall	typeBoolean	R/W	Instructs the engine to create a new System Folder during installation.
pIsParasiteScript	typeBoolean	R/W	Tells the scriptwriter whether they are running as a parasite. See “UpgraderCommandsTool Guide.pdf” for more information about parasites.
pBoxFlagOverwrite	typeLongInteger	R/W	Remaps the machine ID to the value specified. Use this property to allow older Installer scripts that call CheckGestalt rule clause to work on newer unsupported

		machines. See "UpgraderCommandsTool Guide.pdf" for more information about remap IDs.
Element	**Access Form**	**Description**
cEasyFeatureSet	formAbsolutePosition	The easy install installation object. More than one may exist, depending on the number of feature sets implemented by the Installer script.
cCustomFeatureSet	formAbsolutePosition	The custom install installation object. Since there can only be one, access the object using an index of one.
cStatus	formAbsolutePosition	The current status of the document. Since there is only one, access the status object using an index of one.

Status Object

Clients can poll the status object during long installatoins to provide progress information.

Property	Data Type	Mod.	Description
pPercentTaskComplete	cLongInteger	R/O	The percent complete of the current task. The actual value ranges from 0 to 10,000. Therefore, if the task is 64% complete, the value of this property is 6,400.
pOperationType	cType	R/O	The operation type.
pOperationParams	AERecord	R/O	An AERecord containing the parameters for the specific type of operation.

The following table describes the parameters contained in the AERecord returned from the pOperationParams property given the operation type.

Operation Type	Description & Operation Parameters
kTargetAnalysis	Reconciling atoms against the target. 'sttn' Target Volume Name String
kSourceAnalysis	Reconciling atoms against the source disk. 'stsn' Source Volume Name String
kDeleting	Files and resources are being deleted, either at the beginning of an installation or as the result of a removal.
kReadingFile	Reading file data. 'obty' Object Type Type 'obID' Object ID Long

	'obnm' Object Name String 'obds' Opt. Desc. String
kReadingRsrc	Reading resource data. 'obty' Object Type Type 'obID' Object ID Long 'rsty' Rsrc Type Type 'rsID' Rsrc ID Long 'obnm' Object Name String 'obds' Opt. Desc. String
kReadingFont	Reading font data. 'obty' Object Type Type 'obID' Object ID Long 'obnm' Family Name String 'obds' Opt. Desc. String
kReadingOther	Reading some other type of data 'obty' Object Type Type 'obID' Object ID Long 'obnm' Object Name String 'obds' Opt. Desc. String
kWritingFile	Writing file data. 'obty' Object Type Type 'obID' Object ID Long 'obnm' Object Name String 'obds' Opt. Desc. String
kWritingRsrc	Writing resource data. 'obty' Object Type Type 'obID' Object ID Long 'rsty' Rsrc Type Type 'rsID' Rsrc ID Long 'obnm' Object Name String 'obds' Opt. Desc. String
kWritingFont	Writing font data. 'obty' Object Type Type 'obID' Object ID Long 'obnm' Family Name String 'obds' Opt. Desc. String
kWritingOther	Writing some other type of data. 'obty' Object Type Type 'obID' Object ID Long 'obnm' Object Name String 'obds' Opt. Desc. String
kClosingFiles	Closing a target file.
kExecutingExternalAction	Called an Action Atom. 'obty' Object Type Type 'obID' Object ID Long 'obds' Opt. Desc. String

kLookingForSrcDisk	Verifying source disks during process of finding the next source disk. 'stsn' Source Volume Name String
kCancelingOperation	Cleaning up after a cancel request.
kUpdatingTargetDisk	Final target disk updating, most likely adding entries to the desktop database. 'sttn' Target Volume Name String

Easy Feature Set Object

The easy feature set object describes a scriptwriter-defined installation. If more than one object exists, the first object is considered the recommended installation.

Property	Data Type	Mod.	Description
pName	typeChar	R/O	The name of the feature set.
pFeatureSetResult	typeLongInteger	R/O	The error result after firing the easy feature set rule framework.
pFeatureSetMessage	typeChar	R/O	The message text after firing the easy feature set rule framework.
pRestartRequired	typeBoolean	R/O	If TRUE, the scriptwriter has requested that the machine be restarted immediately after the installation. This property will always be FALSE if not installing into the active System Folder. It is up to the client to restart the machine.
pFeatureList	typeLongInteger	R/O	A list of top-level feature IDs contained in this feature set.

Element	Access Form	Description
cFeature	formAbsolutePosition	A top-level feature in this feature set.

Custom Feature Set Object

The custom feature set object provides the client with information about the feature hierarchy from which a user can choose a customized installation or removal.

Property	Data Type	Mod.	Description
pFeatureSetResult	typeLongInteger	R/O	The error result after firing the custom feature set rule framework.
pFeatureSetMessage	typeChar	R/O	The message text after firing the custom feature set rule framework.
pRestartRequired	typeBoolean	R/O	If TRUE, the scriptwriter has requested that the machine be restarted immediately after the installation. This property will

			always be FALSE if not installing into the active System Folder. It is up to the client to restart the machine.
pSubFeatures	typeLongInteger	R/O	A list of top-level feature IDs contained in this feature set.

Element	Access Form	Description
cFeature	formAbsolutePosition	A top-level feature in this feature set.

Feature Object

The feature object describes an individual feature that can be installed or possibly removed.

Property	Data Type	Mod.	Description
pID	typeLongInteger	R/O	The ID of the feature.
pName	typeChar	R/O	The name of the feature.
pNeedsRestartToBeActive	typeBoolean	R/O	This feature requires a restart after installing into the booted System Folder.
pVisible	typeBoolean	R/O	This feature can be selected by the user.
pRemovable	typeBoolean	R/O	This feature can be removed.
pIcon	AERecord	R/O	Contains the feature's icon, as displayed in the info window for the feature. The AERecord returned contains two icons: kIconAndMaskKeyWord ('ICN#') containing a B&W icon and mask, and k8BitIconKeyWord ('icl8') containing the color icon.
pVersion	typeLongInteger	R/O	The feature's version.
pDate	typeLongInteger	R/O	The feature's release date in seconds.
pOnDiskSize	typeLongInteger	R/O	The feature's size in bytes.
pCommentText	typeChar	R/O	The feature's description.
pBasicProperties	AERecord	R/O	A collection of specific feature information in a single property to enhance performance. <To be documented>
pSubFeatures	typeLongInteger	R/O	A list of top-level feature IDs contained in this feature set.

Element	Access Form	Description

cFeature	formAbsolutePosition	Features contained in this feature.

Receiving Events from Installer Engine

Installer Engine sends numerous events back to the client application during certain tasks to keep the client fully informed about the engine's actions. As a client, you should first determine the type of information you need, then understand how to process the information you receive. As is true in many programming situations, the more information you request, the slower the engine can accomplish its primary task.

By default, clients that send an Install or Remove event to Installer Engine are automatically registered to receive basic progress and error events. If you want additional events, such as detailed progress events, or if you want special debugging information you can use the Register Client event to request this information be sent to you. You can send additional Register Client events to change these flags over the course of your session with Installer Engine.

All events sent to the client application have a class of 'inst'. The following table lists the Apple events sent to the client application.

Apple Event	Description
kAEInstallerProgress ('iprg ')	Clients receive basic progress events by default. Detailed progress events can optionally be received by specifying this in the Register Client Apple event.
kAEInstallerError ('ierr')	Clients receiving basic progress events will also receive error events from the engine.
kAEInstallerDebug ('idbg')	Debug events can optionally be received by specifying this in the Register Client Apple event.
kAEInstallerReport ('irpt')	A report requested by the client describing the actions Installer Engine will perform for the given installation or removal. A parameter of the Install or Remove event sent to Installer Engine determines if the client will receive this event.
kAEInstallerAlertRequest ('iadr')	A request from the scriptwriter to display the specified dialog.

Receiving Progress Events

Progress events are divided into basic and detailed events. Basic progress events tell the client when the installation/removal process has finished. Detailed progress events tell the client about individual tasks performed during the installation or removal, such as the name of the file being copied.

Basic progress Event	Description
kOperationFinishedSuccessfully	The operation finished successfully.
kOperationFinishedWithErrors	The operation finished with an error. A separate error is set before this progress event.

Detailed progress events	Description
kTargetAnalysis	Reconciling atoms against the target.
kSourceAnalysis	Reconciling atoms against the source disk.

kDeleting	Files and resources are being deleted, either at the beginning of an installation or as the result of a removal.
kReadingFile	Reading file data.
kReadingRsrc	Reading resource data.
kReadingFont	Reading font data.
kReadingOther	Reading some other type of data
kWritingFile	Writing file data.
kWritingRsrc	Writing resource data.
kWritingFont	Writing font data.
kWritingOther	Writing some other type of data.
kClosingFiles	Closing a target file.
kExecutingExternalAction	Called an Action Atom.
kLookingForSrcDisk	Verifying source files.
kCancelingOperation	Cleaning up after a cancel request.
kUpdatingTargetDisk	Final target disk updating, most likely adding entries to the desktop database.

Receiving Debugging Events

Debugging events are mainly for applications that desire a more intimate relationship with Installer Engine. The Installer Debugger application provided with Installer Engine uses debugging events to monitor Installer Engine remotely for testing and development purposes.

A debugging event has an ID of kAEDeregisterClient ('idgc'). The following table describes the parameters.

Parameter	Data Type	Description
'dbID'	typeLongInteger	The ID of the debugging action that occurred.

Additional parameters may be included depending on the action type. The following table describes these actions.

Debugging Action	Description
kDocumentOpened (0)	A new document was just opened.
kDocumentClosed (1)	A document is about to be closed.
kApplicationQuit (2)	A Installer Engine is in the process of quitting.
kEngineAtion (3)	An action to be performed by Installer Engine. The event contains six additional parameters that further describe the action: kEngineActionClass 'eaci' typeLongInteger kEngineActionID 'eaid' typeLongInteger

kEngineActionParam0 'eap0' Depends on action ID
kEngineActionParam1 'eap1' Depends on action ID
kEngineActionParam2 'eap2' Depends on action ID
kEngineActionParam3 'eap3' Depends on action ID

See “ActionHandlerHeader.h” for information about the defined action classes and IDs sent by Installer Engine.

Receiving Error Events

Clients receiving basic progress events will also receive all error events. When something goes wrong during an installation, an error event is sent first to the client, then the kOperationFinishedWithErrors basic progress event is sent. The following table describes the paramters.

Parameter	Data Type	Description
keyErrorNumber ('errn')	typeLongInteger	The error number.
keyErrorString ('errs')	typeChar	A string desribing the error.
keyRollBackFlag ('rbfl')	typeBoolean	True if the installation was canceled and all original files were restored.

See the chapter “Runtime Issues” for a description of the errors returned by Installer Engine.

Receiving a Report

Requesting an installation report when sending an Install or Remove event to Installer Engine causes a series of report events will be sent back to the client application. Since the report may be large, and Apple events have a size limit, multiple events will be sent with the last event containing an empty list.

A report event has an ID of kAEInstallerReport (‘irpt’). The following table describes the paramters.

Parameter	Data Type	Description
kReportType ('rptt')	typeInteger	The type of report. This parameter will be one of two values: kPreInstallation (0) or kPreRemoval (2).

kReportList ('rptl')	AEList of ‘rptr’	A list of records of type ReportItem.
		
		`typedef struct`
		`{`
		`  SInt16 fActionClassID;`
		`  SInt16 fActionIdentifier;`
		`  SInt16 fTgtVolRefNum;`
		`  SInt32 fTgtDirID;`
		`  Str255 fTgtFilePath;`
		`}  ReportItem;`

The target file being acted upon is specified in the fTgtVolRefNum , fTgtDirID, and fTgtFilePath fields. You can use these fields to create an FSSpec to the file. The following

table describes the possible values passed in the fActionClassID and fActionIdentifier fields of the ReportItem record.

Action Class ID	Action Identifier	Description
kFileProgressAction (5800)		
	kFileCopiedID (5801)	The file will be copied to the location specified.
	kFileNotCopiedNewerLeftID (5803)	The existing file will not be updated because the existing file is newer than the file we would be copying.
	kFileNotCopiedLockedID (5804)	The existing file will not be updated because it is locked.
	kFileUpdatedID (5805)	The existing file will be updated.
	kFileRemovedID (5808)	The existing file will be updated.
kRsrcProgressAction (5900)		
	kRsrcCopiedID (5901)	A resource will be copied to the specified file. You aren't told which resource is copied, but only that a resource will be copied. If you need to know the resource type and ID, then you should accept debugging events instead.
	kRsrcUpdatedID (5906)	A resource will be update in the specified file.
	kRsrcRemovedID (5911)	A resource will be removed from the specified file.
kFontProgressAction (6000)		
	kFontCopiedID (6001)	A font resource will be copied to the file specified. You aren't told which resource is copied, but only that a resource will be copied. If you need to know the resource type and ID, then you should accept debugging events instead.
	kFontUpdatedID (6003)	A font resource will be update in the specified file.
	kFontRemovedID (6008)	A font resource will be removed from the specified file.
kFolderProgressAction (6100)		

kFolderCopiedID (6101)	The contents of the source folder will be merged with the specified folder.

kResMergeProgressAction (6200)

kResMergeCopiedID (6201)	All resources contained in the source file be copied to the specified file.

Receiving a Message Alert Display Request

Scriptwriters can no longer interact directly with the user via windows or dialog because Installer Engine is a background-only application. To fix this limitation, Installer Engine provides the callback routine PresentMessageAlert to handle simple dialog presentation and interaction from within code resources such as action atoms. When the scriptwriter calls PresentMessageAlert, Installer Engine sends an Apple event to the client application requesting it to display the specified dialog. If the client does not support displaying the dialog, it returns an error number and the scriptwriter handles this refusal appropriately.

The message alert event has an ID of kAEInstallerAlertRequest (‘iadr’). The following table describes the paramters.

Parameter	Data Type	Description
keyAlertRequestDITLID ('idid')	typeLongInteger	The ID of a ‘DITL’ resource supplied by the client application. Most Installer scripts use the IDs defined by the Upgrader shell application. See *Upgrader Tech. Guide* for more information.
keyAlertRequestMsgText ('iamt')	typeChar	The text message to display in the dialog.
keyAlertRequestCancelBut ('icbu')	typeLongInteger	The index number of the cancel button to map to Command-period. Pass zero if no cancel button is desired.
keyAlertRequestDefaultBut ('idbu')	typeLongInteger	The index number of the default button to outline and map to the return key. Pass zero if no default button is desired.

Installer Engine expects the button number clicked by the user returned in the direct object parameter (typeLongInteger) of the reply Apple event. If the client application could not display the dialog or is inappropriate at this time, a value of –1 should be returned in the direct object parameter.

Responsibilities of a Client

As the client application, you assume responsibility for certain aspects of the installation process that involve user interaction, and therefore cannot be performed by the engine. Evaluate each of the following areas to determine if your client application must accommodate the situation.

Replacing Newer Files

Scriptwriters have an option to force an existing file to be downgraded to an older version during an installation. Apple suggests that scriptwriters only force downgrading when compatibility across set of files is required. The majority of existing scripts preserve the newer file, which seems to meet the expectations of most users.

The client can control whether Installer Engine silently downgrades the newer files, or reports each file as a separate error and thus canceling the installation. The cDocument property pReplaceNewerFiles provides this option. Set the pReplaceNewerFiles property to true to reduce the instances of errors stopping the installation.

Replacing Locked Files

Installer Engine currently prevents an installation from continuing that will replace locked files. Installer 4.0 and later scripts have an option to allow the scriptwriter to ignore the Finder's locked flag for a specific file and silently replace the file.

Since this is another annoying error that would stop remote installations from completing, an option can be set to silently replace any locked file. The cDocument property pReplaceLockedFiles provides this option. Set the pReplaceLockedFiles property to true to reduce the instances of errors stopping the installation.

Quitting Applications and Forcing Restarts

Scriptwriters can force a restart after a live installation by setting a flag in packages that touch live files, such as the System file. Since Installer Engine contains no code for quitting applications and restarting the machine, the client is responsible for performing these tasks. The client should query the pRestartRequired property of the cEasyFeatureSet object before initiating an installation so an appropriate user experience can be presented.

To reduce the number of restarts required, Installer Engine supports installing multiple products, one after another, without quitting Installer Engine. This is handy when installing system software, which may require various additional system software installations on top of a base installation.

Source Disk Limitations

Installer Engine does not support installing from multiple source disks. It assumes all necessary source disks are mounted or stored as folders on a mounted volume. If for some reason a folder or file cannot be found, an error will be registered and the installation will be canceled.

By default, Installer Engine looks in four places to find the source folder: the script's folder, the script's parent folder, Installer Engine's folder and finally Installer Engine's parent folder. Clients can specify another folder by setting the pSourceLocation property in the cDocument object before starting the installation.

Handling Parasite Installer Scripts

A callback routine allows scriptwriters to check if the Installer script is being installed as parasite. Parasites are a type of invisible software component installed by the Upgrader application. Unless your client application mimics the actions of Upgrader, you don't need to do anything special because Installer Engine defaults to false unless told otherwise.

To tell Installer Engine that the Installer script is a parasite, set the pIsParasiteScript property in the document object to true immediately after opening the Installer script document.

Remapping Machine IDs

A feature in Installer Engine allows the client application to trick the Installer script into thinking it's running on a different machine. This is necessary when the scriptwriter would rather add a parasite Installer script to an older Installer script instead of reopening the older Installer script. The client application needs to recognize that it is running on a machine that is not supported by the older Installer script, and give Installer Engine another Gestalt machine ID to use instead.

To remap the ID, set the pBoxFlagOverwrite property in the document object to the older machine ID immediately after opening the Installer script document.

AppleScript Example

Writing an AppleScript to perform an installation using Installer Engine requires the AppleScript writer to write code to wait for the installation to complete after sending the "install" Apple event. Examine both examples to determine which method of waiting is best for your situation.

AppleScript Example Using Polling Method

For simple automation of an installation or removal from within Script Editor, the example in Listing 8-1 demonstrates an easy way to wait for the installation to complete by polling the status object. This method allows the AppleScript code to be executed from within Script Editor, but doesn't display any error messages if the installation does not complete successfully.

Listing 8-1 Example AppleScript code using polling technique

```
tell application "Installer Engine"

    open file "Mac OS 8.1:System Software:Install Mac OS 8.1"
    set System Target of Document 1 to file "Macintosh HD:"
    Install

    set percentage to 0
    repeat while (percentage is not 10000)
        copy Percent Task Complete of Status 1 of Document 1 to percentage
    end repeat

end tell
```

AppleScript Example Using Event Handlers

Using a progress event handler to signal when the installation process is complete reduces the overhead of polling for the information. Providing an additional error handler will allow the AppleScript writers to display the error that stopped the installation process. Unlike the polling example, this AppleScript must be saved and run as an application; otherwise, Script Editor will receive the progress and error events instead of your AppleScript handlers.

Listing 8-2 Example AppleScript code using event handlers

```
on «event instierr» given «class errn»:errNum, «class errs»:errMsg --, «class
EMS1»:errMsg1, «class EMS2»:errMsg2, «class EMS3»:errMsg3
    display dialog errMsg
end «event instierr»

on «event instiprg» given «class opID»:operationNum --, «class EMS0»:errMsg0, «class
EMS1»:errMsg1, «class EMS2»:errMsg2, «class EMS3»:errMsg3
    global doneYet
    if operationNum = 22 then
        display dialog "Operation was successful."
        set doneYet to true
    else if operationNum = 23 then
        display dialog "Operation was canceled."
        set doneYet to true
    end if
end «event instiprg»

on run
    tell application "Installer Engine"
        global doneYet
        set doneYet to false

        open file "Mac OS 8.1:System Software:Install Mac OS 8.1"
        set system target of document 1 to file "Macintosh HD:"
        install feature names {"Core System Software"}

        repeat while not doneYet
        end repeat

    end tell
end run
```