CHAPTER 21

Using the Installation Framework

CWCS supplies several tools for application installation, uninstallation, and patching. Because each platform has its own set of standards, formats, and issues, you will need different tools. But the basic installation concepts are similar.

Note Cisco encourages developers to leverage the installation framework discussed in this chapter. The tools for user interface and installation integration are free and will save your team time.

The installation framework allows you to consider dependencies and features such as uninstallation, which makes it easier for your package to work like other network management packages. The installation framework can help ensure that prerequisites such as version dependencies are set and followed.

Cisco recommends that you use both the build environment and the installation framework. The build environment is tuned to enforce Cisco procedures and policies. The installation framework facilitates a common look-and-feel and the required CWCS bundle behavior. However, the installation framework does *not* require using the build environment.

The following topics describe the installation framework and associated processes:

- About the Installation Framework
- Getting Started with the Installation Framework
- Using the Installation Framework
- Windows Installation Reference
- Solaris Installation Reference

For information on installing CiscoWorks Common Services itself, refer to the "Installing CWCS" section on page 5-2.

About the Installation Framework

This section discusses the following topics:

- What's New in This Release
- Understanding the CWCS Installation Framework
- Understanding Installation Team Responsibilities
- Understanding Developer Responsibilities

What's New in This Release

New or changed features in this release of the CWCS installation framework include:

- Express mode is no longer supported. The only installation modes are Typical and Custom.
- Remote upgrade is no longer supported. This has been replaced by the CWCS migration framework.
- The installation framework is now compatible with the new CWCS licensing framework, vastly simplifying the work applications need to do (see the "Providing Licensing Information During Installation" section on page 21-22).
- Advice on reducing installation time has been added (see the "Reducing Windows Installation Time" section on page 21-36).
- Solaris installation now supports limited workflow customization (see the "Customizing the Installation Workflow on Solaris" section on page 21-104)
- The new command pkgchk permits package verification before or after Solaris installs (see the "Verifying Packages on Solaris" section on page 21-105).

Understanding the CWCS Installation Framework

The CiscoWorks Common Services (CWCS) runtime environment provides shared functionality for applications when they are running. It provides services such as scheduling, process management, and database storage and retrieval so that your applications can perform their tasks. The CWCS installation framework provides shared functionality for individual applications so that they can install on different kinds of systems with similar interfaces and consistent install semantics.

The goals of the installation framework are:

- To isolate the installation process and tools (as much as possible) from the application so the application only needs to worry about application-specific install issues.
- To allow applications to specify hardware and software requirements as well as dependencies, regardless of the target operating system.
- To support the orderly uninstallation of applications.
- To facilitate a common user experience for all CiscoWorks products.
- To allow the user to install the package easily, regardless of the target operating system or application being installed.
- To update package file ownership settings during the build/installation on Solaris.

The installation framework provides an installer, an uninstaller, and a set of functions that facilitate installation-specific tasks.

The installation team and the developer have separate roles to play in the process of adding packages for installation:

- The installation team is responsible for ensuring that the main installation script and build tools (the installation framework) are available and working (see the "Understanding Installation Team Responsibilities" section on page 21-3).
- The developer is responsible for specifying what objects should be installed, where they should be installed, and how they should be installed (see the "Understanding Developer Responsibilities" section on page 21-3).

Understanding Installation Team Responsibilities

The installation team is responsible for the main installation script. This script is the same for CiscoWorks, Resource Manager Essentials, Campus Manager, and other CWCS-based products. It does not change (on Windows platforms, the main installation script can be customized; for details, see the "Customizing the Installation Workflow for Windows" section on page 21-76). It is important to use the installation framework to allow for the following:

- Component sharing: Enables several packages within the product to share functionality without installing packages more than once.
- Dependencies: Includes source and target, version, backward compatibility, and uninstallation.
- Optional components: installation of optional components based on requirements or dependencies.
- Development: Allows individual business units or third-party developers to produce components.
- Delivery: Allows individual business units or third-party developers to release their products as part of a major or minor release or drop-in.
- Maintenance: Provides for backward compatibility, upgrades, and patches that are consistent.
- Security: Allows CiscoWorks applications to use the same file ownerships and permissions schema as CWCS (see the "Understanding and Implementing the casuser" section on page 21-21).

The installation team is also responsible for the tools that create the installable CD image, which are discussed in the "Getting Started with the Installation Framework" section on page 21-4. Developers outside of NMTG can use their own build tools and build environment to produce their executable files, but should use the installation tools to build CDs. NMTG developers should use internal build and installation tools.

For assistance with running the installation framework software, reporting problems, or questions about the software:

- Refer to the Installation Framework team web site: https://mco.cisco.com/ubiapps/portal/servlet/EEngPortalDispatcher?EVENT=ProjectPortalDisplayEvent&portal_id=1975
- Contact: cmf-install-dev@cisco.com

Related Topics

See the:

- "Understanding Developer Responsibilities" section on page 21-3.
- "Understanding the CWCS Installation Framework" section on page 21-2.

Understanding Developer Responsibilities

The CWCS installation framework is provided by the installation team, but can be used in different ways depending on your requirements:

- If you are a developer working outside NMTG (either in another business unit or as a third-party partner), you should use the installation framework to enable better integration, improve look and feel, and take advantage of features such as uninstallation.
- If you are a developer working inside NMTG, in addition to using the installation framework you must use the automated build processes to create protopackages. Your build process has the build tools to automatically create protopackages.

Related Topics

See the "Understanding Installation Team Responsibilities" section on page 21-3.

Getting Started with the Installation Framework

The Installation Framework is supported on Windows and Solaris platforms. It follows the server platform-support requirements for CWCS, but can be customized for other Windows and Solaris platforms. For other platform enquiries, contact the installation team (cmf-install-dev@cisco.com).

Main installation script and build tools for the CWCS installation framework are available from the CWCS 3.0 SDK Portal at https://mco.cisco.com/ubiapps/portal/go.jsp?portal_id=2537.

This section discusses the following topics:

- Third-Party Tools for Installation Framework
- Understanding Install Component and Image Structures
- Preparing Installation Protopackages

Third-Party Tools for Installation Framework

The following tables describe the required third-party tools for Windows and Solaris:

Table 21-1 Third-Party Tools for Windows

Name	Description
InstallShield 5.53 Professional only	Installs packages and files. Purchase from InstallShield Software Corporation. The Maintenance Pack 3 for the InstallShield 5.5 is required.
InstallShield PackageForTheWeb version 4.x	Packages CD image into the self-extracting .EXE file. Download from InstallShield Software Corporation. See http://www.installshield.com.
MKS Toolkit 6.1 or higher	Builds an image from protopackages. Purchase from the Mortice Kern Systems Inc., Tel: (519) 884-2251, http://www.mks.com

Table 21-2 Third-Party Tools for Solaris

Name	Description
Perl version 5.5 or higher	Builds an image from protopackages. Shareware/public domain software located at http://www.perl.com/pub.

Understanding Install Component and Image Structures

CWCS software is divided into a set of components that reflect the logical structure of the software. In this document, the following terms are used to describe the package structure:

- package—The smallest piece of software processed by the installation framework. Packages are defined by a set of properties. All runtime files belong to packages.

- installable unit—A group of packages that are always installed or uninstalled simultaneously. A package can only belong to one installable unit (or can belong to no installable unit). An installable unit is also defined by a set of properties; on Solaris it cannot directly include runtime files.
- suite—A group of packages or installable units that define a product or an application, usually as required by marketing.
- component—A package, installable unit or suite.
- protopackage—A protopackage is a tar file that contains a component of the product. This component has a name, version, and other properties. It also contains the runtime and installation-related information. A protopackage is used to prepare an input to the installation framework. Typically, protopackages are created automatically by the build tools after compilation, linking, etc.
- tag—Protopackages are identified by a tag. This tag is the short internal name, which is the same as the first word in the name of the protopackage file. For example, the cmf.runtime.tar protopackage has this tag: cmf.
- image—A set of files that contains one or more products ready for installation. An image can be directly burnt on a CD. It contains the installer as well as all product components. Each image contains the Table of Contents file (disk.toc) that specifies the metadata about the image, as well as some instructions for the installer.

The following topics describe the process to follow in building installable images, and the package components and image structures:

- Building an Installable Image
- Selecting Package Names
- Specifying Package Properties
- Understanding the Package Properties File
- Understanding Suite Properties
- Creating the Table of Contents

Building an Installable Image

Use the following process to build an installable image:

Step 1 Select the package name (see the "Selecting Package Names" section on page 21-6) .

Step 2 Specify properties for packaging (see the "Specifying Package Properties" section on page 21-6).

Step 3 Write scripts to specify installation requirements and enforce constraints:

- For Windows, write InstallShield scripts (see the "Writing Windows Scripts" section on page 21-33).
- For Solaris, write Bourne shell scripts (see the "Writing Solaris Scripts" section on page 21-90).

Step 4 Prepare the Table of Contents (see the "Creating the Table of Contents" section on page 21-11).

Step 5 Create protopackages that include the runtime files, package properties, scripts, and Table of Contents created in the previous steps (see the "Preparing Installation Protopackages" section on page 21-19).

Step 6 Build and debug CDs (see the "Using Solaris Build Tools" section on page 21-103).

Selecting Package Names

A package name, or *tag*, is a short internal name that is used programmatically. Normally it is not presented to the end user.

Windows Package Names

For Windows platforms, the package name must match the name of .pkgpr file, for example ut.pkgpr must have PKG=ut. The following table describes the Windows package name requirements.

Table 21-3 Windows Package Name Requirements

Requirements	Description
Naming Conventions	Eight character limitation, CSCO prefix is not required. For example, ut.
Package name must match pkgpr files	Required. Example, ut.pkgpr.
Names must be unique	Required, but only within CiscoWorks products. It will not collide with non-CiscoWorks products.

Solaris Package Names

For Solaris, the package name must begin with CSCO followed by up to five characters. The following table describes the Solaris package name requirements.

Table 21-4 Solaris Package Name Requirements

Requirements	Description
Naming Conventions	CSCO prefix is required, with no more than nine characters total. For example, CSCOjrm.
Package name must match pkgpr files	Not required.
Names must be unique	Required. Must be unique across all products on this platform.

Specifying Package Properties

Use the package properties file (*tag*.pkgpr) to specify the component properties for each package. The package properties file contains name-value pairs for the platforms your package supports and specifies the properties for each platform.

Any platform-specific name value pairs should be specified in the package properties file using the following line immediately before the pairs appear in the file:

```
PLATFORM_NAME:
```

Note Remember to always use a colon to separate any platform names or to complete the end of a line.

The platform names are represented as:

- NT: for Windows
- SOL: for Solaris

For example, any Solaris-specific name value pairs in the package property file must have the line:

```
SOL:
```

Several of them can be combined as follows:

```
SOL:NT:
```

Understanding the Package Properties File

The package properties file (<tag>.pkgpr) include several groups of properties. The following topics describe these property groups:

- Developer-Specified Properties
- Appended and Generated Properties
- Solaris-Specific Properties

Developer-Specified Properties

Table 21-5 show the package properties that developers must specify for both Windows and Solaris platforms. The image build tools will transfer these name-value pairs into the <tag>.info files. Finally, the installation process copies them into the target system.

Table 21-5 <tag>.pkgpr Files Name-Value Pairs

Name	Format	Description	Required
PKG	Solaris: CSCOxxxxx Windows: up to eight characters	Package tag. A short internal name, used programmatically. Normally invisible to the end user. For Solaris, it must begin with CSCO followed by a maximum of five characters. For Windows, it must match the name of the .pkgpr file. For example, ut.pkgpr must have PKG=ut.	Yes
NAME	String, up to 40 characters	One-line name to be presented to the end user whenever components are listed.	Yes
DESC	String	Short description (up to 1000 characters).	Yes
VERSION	X.Y, where X and Y are numbers	Version, major and minor	Yes
PATCHVER	Number	Patch level	0 by default
DEPENDS	String	Comma-separated list of components this one depends on, each in the form of *pkgTag major.minor.patchlevel*, where *pkgTag* is the package tag; *major*, *minor*, and *patchlevel* specify the version of dependency target. If a specific version of dependency target is not required, then you can omit *major.minor.patchver.* Correct: DEPENDS=cmf, xrts 1.2.1, ani Not correct: DEPENDS=xrts 1.2 (It should be DEPENDS=xrts 1.2.0.)	No
ROOTDIR	String	Alias for root directory. Default value *NMSROOT.* [1]	No

Table 21-5 <tag>.pkgpr Files Name-Value Pairs (continued)

Name	Format	Description	Required
PACKAGES	String	Space-separated list of packages that belong to this installable unit. This property must be specified for installable units only.	Yes, f or installable units only.
BACKVER	X.Y.Z (where X, Y, Z are numbers)	The *major.minor.patchlevel*, where *major*, *minor*, *patchlevel* specify the version. If you do not use all three version numbers, use no version numbering. This version is backward compatible with all versions starting from the one specified by this parameter. Example: `VERSION=3.2, PATCHLEVEL=3, BACKVER=2.4.0.` If any other package depends on version 2.6 of this package, it can be installed.	No
OPTIONAL	Y	Specifies that a package is optional. An optional package can be dropped if dependencies for it cannot be resolved. It is assumed that installable unit is operational with or without optional packages.	No
SIZE	Number	Space, in megabytes, required for this package. The installer calculates a package's footprint automatically. This property provides for override. Windows calculates this automatically.	Yes, for Solaris only.
REC_RAM REC_SWAP REC_DISK REC_CPU		These properties can be set for packages or installable units but are normally set for Suites only.	No
CMFSERVICES	String	Comma-separated list used to register for specific CiscoWorks service bundle use. Possible values: System, Network, Core.	No
UNINSTALL_REBOOT	Windows only: Y or y	Indicates that the system needs to be rebooted at the end of uninstallation (if this package is uninstalled).	No

1. The installer maintains the list of root directories for all components. The value of ROOTDIR property is an alias, which can be used in installation hooks. See the "Locating the Root Directory Path Name" section on page 21-54 for more information.

Appended and Generated Properties

Other properties are also present in the package properties file. These are appended to the file during the build process or generated by the installer:

Note The properties in Table 21-6 and Table 21-7 appear in the <tag>.info files, but *not* in the <tag>.pkgpr files.

- Table 21-6 describes the properties appended to the properties package file by the build process.
- Table 21-7 describes the properties generated by the installer. Platform-specific requirements are noted where applicable.

Table 21-6 Properties Appended During the Build Process

Name	Format	Description
BUILD_ID	String	Build ID in the form platform_project_date_type. For example: NT_CMF_RIGEL_19990408_0126_daily. This ID id is generated by **make protopackage**[1] by combining the values of PROP_ID and PROP_WHEN from the .bprops file.
BUILD_TSTAMP	Number	Timestamp, generated from PROP_TIMESTAMP in the .bprops file.
ALIAS	String	Contains the names following the PKG/SUITE property for each platform. Contents written to the .info file. For example, dmgt.info/CSCOmd.info both contain the new property ALIAS=dmgt CSCOmd. Using a space field separator, you can determine if a .info file matches the package by examining the ALIAS property.

1. **make protopackage** is specific to the NMTG build environment.

Table 21-7 Properties Generated by the Installer

Name	Format	Description
Date	String	Date and time of installation.
PREV_VERSION	See VERSION	Version that has been installed before this one.
PREV_PATCHVER	See PATCHVER	Patch level that has been installed before this one.
I_MODE (UNIX only)	String	NEW, REINSTALL, DOWNGRADE, PATCH, or UPGRADE.
OS (UNIX only)	String	SOL, HPUX, or AIX.

Solaris-Specific Properties

Table 21-8 describes Solaris properties that are not generated automatically, *but that are required*. Add these lines to your pkgpr file.

Table 21-8 Solaris-Specific Properties

Property Name	Definition	Example
CATEGORY	Type of package	`CATEGORY=application`
ARCH	Architecture package supports	`ARCH=sparc`
VENDOR	Who created the package	`VENDOR=Cisco Systems, Inc.`
CLASSES	For character abbreviation	`CLASSES=TBD`
BASEDIR	Top-level install directory.	`BASEDIR=/opt`

The information about prototype.header properties listed in Table 21-9 is provided for developers who are not familiar with Solaris packaging tools. The prototype.header file required for Solaris packaging is generated automatically and is not to be maintained by developers. If you create your own prototype.header file, we will overwrite it.

Note The properties in Table 21-9 should *not* be specified manually by developers.

Note Developers can define their own properties in the *tag*.pkgpr file for their own use.

Table 21-9 Properties for prototype.header

Name	Format	Description	Required
IU_NAME	String, up to 256 characters	One-line name presented to the end user. Used whenever components are listed.	Yes
IU_VERSION	X.Y, where X and Y are numbers	Version, major and minor.	Yes
IU_DESC	String	Short description—up to 1000 characters.	No
IU_PATCHVER	Number	Patch level.	No
DEPENDS	String	Comma-separated list of components this one depends on, each in the form of *pkgTag major.minor.patchlevel*, where *pkgTag* is the package tag; *major*, *minor*, *patchlevel* specify the version of the dependency target.	No

Name	Format	Description
I_MODE	String	NEW, REINSTALL, DOWNGRADE, PATCH, or UPGRADE
NMSROOT	String	Path name of target directory
SETUPDIR	String	Path name from which the CD installation is running

Understanding Suite Properties

Suite properties describe different bundles of installable units and/or packages. Table 21-10 describes the suite properties for both Windows and Solaris.

Table 21-10 Suite Properties

Name	Format	Description
SUITE	Up to eight characters	Suite tag. Tag is short internal name, which is used programmatically. Normally it is not presented to the end user.
NAME	String, up to 40 characters	One-line name to be presented to the end user.
DESC	String	Short description, up to 1000 characters.
VERSTR	String	Version string to be presented to end user. For example, "Service Pack 3."
DEPENDS	String	Comma-separated list of components this one depends on, each in the form of *pkgTag major.minor.patchlevel*, where pkgTag is the package tag; *major*, *minor*, and *patchlevel* specify the version of dependency target. If a specific version of dependency target is not required, then *major.minor.patchver* can be omitted. Correct: `DEPENDS=cmf, xrts 1.2.1, ani` Incorrect: `DEPENDS=xrts 1.2` (It should be `DEPENDS=xrts 1.2.0`.)
TAGS	String	Space separated list of packages or installable units, which belong to this suite.

Table 21-10 Suite Properties (continued)

Name	Format	Description
REC_RAM REC_SWAP	Number	Specifies the recommended size of RAM (in MB) and swap/paging file size (in MB) for this suite. The install process collects these requirements from all components, finds the maximum and verifies if the server has enough RAM and swap. This process displays a warning if these requirements are not satisfied. It is warning only. Installation will not abort.[1]
REC_CPU	Number	(Windows only.) Specifies the recommended CPU speed in MHz. The install process collects these requirements from all components, finds the maximum and verifies if the server has enough RAM and swap. The install process displays a warning if these requirements are not satisfied. It is only a warning. Installation will not abort.[1]
SSL_COMPLIANT	String. Possible values: Yes or No	Specifies whether the suite is SSL compliant. The installation process detects whether the suites are SSL compliant. The installation program does not allow the installation of the suite if the underlying Ciscoworks installation is SSL enabled. This field is also used by other components, such as Daemon (Process) Manager, and Enable/Disable SSL. The installation framework, daemon manager and other modules of CWCS check the SSL_COMPLIANT field in *.info files under *NMSROOT*/setup, only if the info files have a SUITE field. In other words, CWCS detects SSL compliance only in SUITE level information files. If an application cannot add the SUITE tag in info files, you can use the others directory under *NMSROOT*/setup. CWCS checks for SSL-compliance in all information files in *NMSROOT*/setup/others directory, irrespective of whether it contains the SUITE tag or not. The application should store such info files under *NMSROOT*/setup/others directory. **Note** Other CWCS modules uses the info files under *NMSROOT*/setup. So the applications can store the info files in both directories, to make use of the appropriate CWCS function. The applications and versions page will use the SSL_COMPLIANT value from info files from both directories - *NMSROOT*/setup *NMSROOT*/setup/others.

1. On PCs, the CPU speed and RAM size are often reported incorrectly by MS Windows. For example, HP Vectra reports 1 MB RAM less than it actually has. For this reason, installation will consider RAM and CPU speed sufficient if the server has 5 MB less than required.

Creating the Table of Contents

The table of contents is specified by disk.toc, an ASCII file that contains:

- The contents of the CD
- Components available on this CD
- Defaults for installation
- Release-specific information (release name, etc.)

The following is an example of a table of contents file:

```
[RELEASE]
NAME=Test Application with CiscoWorks Common Services 3.0
PRODUCT_NAME=CiscoWorks
```

```
SHORT_PRODUCT_NAME=CiscoWorks
VERSTR=1.0

[COMPONENTS]
TAGS=cdone, CSCOmyApp
UNINSTALLABLE=cdone, CSCOmyApp
VISIBLE=cdone, CSCOmyApp
DEFAULT=cdone, CSCOmyApp

[ADVANCED_CHOICE_1]
ADVANCED_CHOICE_1_CONDITION=TRUE
ADVANCED_CHOICE_1_TYPE=NONE
ADVANCED_CHOICE_1_DEFAULT=1
ADVANCED_CHOICE_1_1_TEXT=Everything
ADVANCED_CHOICE_1_1_TAGS=cdone, CSCOmyApp
```

The table of contents consists of the following sections. Each section defines one or more parameters in the form of name=value pairs.

- RELEASE—Contains the name, description, and version string for the release (see the "Creating the RELEASE Section" section on page 21-12).
- COMPONENTS—Specifies additional properties for components (suites, installable units, and packages) available on the CD (see the "Creating the COMPONENTS Section" section on page 21-13). This section is mandatory.
- Optional override sections—Each section has a name, which is a valid component tag. Parameters in this section override those specified as component properties or provide additional properties (see the "Creating Overrides Sections" section on page 21-14).
- ADVANCED_CHOICE—Each section defines a component choice scenario. There can be more than one scenario; each section describes one scenario (see the "Creating Advanced Component Sections" section on page 21-14).
- PROMPT_INSTALL_TYPE—**Solaris only:** Specifies the installation mode (Typical or Custom) (see the "Creating the PROMPT_INSTALL_TYPE Section" section on page 21-16).
- INSTALL_TYPE_SELECTION—**Windows only:** Specifies the installation mode (Typical or Custom) (see the "Creating the INSTALL_TYPE_SELECTION Section" section on page 21-16).
- BUILD—Automatically generated at packaging time (see the "Understanding the BUILD Section" section on page 21-17).
- REQUIRED—Specifies the products that are required for this installation (see the "Creating the REQUIRED section" section on page 21-18).

Related Topics

See the:

- "Preparing Installation Protopackages" section on page 21-19.
- "Step 3: Prepare the Make Image on Solaris" section on page 21-103.
- "Customizing the Installation Workflow for Windows" section on page 21-76.
- "Solaris Getting Started Example" section on page 21-106.

Creating the RELEASE Section

The RELEASE section of the table of contents file, disk.toc, contains the properties of this release. This data is displayed to the end user during installation and is included in history and installation log files. It is not included in the component database.

Table 21-11 Table of Contents File—Release Section

Name	Format	Description
NAME	String, less than 20 characters	Name of this release. Included in the welcome dialog.
DESC	String, less than 100 characters	Description. Can provide additional information about the CD.
VERSTR	String	Version string. This can be something like "2.5 Eval" or "Maintenance release." This is displayed next to the NAME and can provide additional information about CD.
README	String.	Optional for Windows only. Path name for this file is displayed at the end of the installation if the end user elects to display the readme file. For example, README=*NMSROOT*:\htdocs\help\netc\ref_config.html

Creating the COMPONENTS Section

The COMPONENTS section in the table of contents file (disk.toc) lists the major components available on this CD and the user information about these components. This section is mandatory.

Table 21-12 Table of Contents File — COMPONENTS Section

Name	Format	Description
TAGS	Comma-separated list	Comma-separated list of components. Each value is a tag of component. Components are defined by corresponding property set (see the "Specifying Package Properties" section on page 21-6" and the "Creating the Table of Contents" section on page 21-11). The installer starts processing the CD from this list. For each component listed, it looks for the subcomponents defined by the PACKAGES property. It is not necessary to include all packages/installable units in this list. It requires only the roots of component hierarchy. Mutually exclusive with PATCH_TAGS.
PATCH_TAGS	Comma-separated list	Comma-separated list of components. Each value is a tag of component. Components are defined by a corresponding property set (see the "Specifying Package Properties" section on page 21-6" and the "Creating the Table of Contents" section on page 21-11). This list defines the top level components available on CD and also turns on the patch mode of installation. This information is used instead of the TAGS for patching as described in "Handling Patches" section on page 21-27.
VISIBLE	Comma-separated list	Comma-separated list of components, that are exposed to the end user during installation. This is the short form to specify VISIBLE=Y for several components. The names of these components will show up in the Component Selection dialog (for custom installation) and in the Confirm dialog. If the component has both CHOICE and VISIBLE properties set to Y, the end user will be allowed to select or deselect this component during installation.

Table 21-12 Table of Contents File — COMPONENTS Section (continued)

DEFAULT	Comma-separated list	Comma-separated list of component tags, which are selected for installation by default. The value can be ALL. Short form to specify DEFAULT=Y for several components.
CHOICE	Comma-separated list	Comma-separated list of components for user to select. These components can be turned on or off in the Component Selection dialog. A component can be omitted during installation in one of the following cases: • Component has both VISIBLE and CHOICE set to Y and turned off by end user during installation. • Dependencies for component cannot be satisfied and the OPTIONAL property for that component is set to Y. (For more details, see Table 21-5 on page 21-7.) **Note** This property provides for trivial component selection. For advance component selection, see the "Creating Advanced Component Sections" section on page 21-14. The CHOICE property is ignored if disk.toc contains one or more ADVANCED_CHOICE sections.
UNINSTALLABLE	Comma-separated list	List of component tags, which can be uninstalled later. **Windows only:** The Uninstall *componentName* option will be created in the dialog displayed by the uninstallation process.
OVERRIDES	Comma-separated list	List of component tags, for which the *tag* sections are available in this table of contents.

Creating Overrides Sections

Each override section has a name that is a valid component tag. Parameters in this section either override those specified as component properties or provide additional properties.

The name of each section is the same as the component tag. It can be the name of the suite, installable unit, or package for which additional properties or override properties are specified in the *tag*.info or pkginfo file. The section name, *tag,* must be included in the OVERRIDES property in the COMPONENTS section.

Creating Advanced Component Sections

The Installation Framework supports complex component choice scenarios by specifying one or more ADVANCED_CHOICE_x sections. Each section defines a component choice scenario. There can be more than one scenario; each section describes one scenario. The section name must use this naming convention:

```
ADVANCED_CHOICE_x
```

where *x* =1, 2, ...

Specify these sections sequentially starting from 1. Install attempts to load these sections beginning with ADVANCED_CHOICE_1, ADVANCED_CHOICE_2, and so on, and stops when the next section is not available.

For each section, the installer checks the condition parameter (see Table 21-13). If the condition fails, the installer looks for the next section. If the condition is satisfied, the installer starts using that section and ignores the remaining sections.

Table 21-13 Table of Contents File — ADVANCED COMPONENT Sections

Name	Format	Description
ADVANCED_ CHOICE_x_CONDITION	TRUE or a comma- or space-separated list of constructs *tag* [.version[-version]]. Version must be in a form minor.major.patchver, and all three of them are required or can be dropped at once with the separator. Examples: • cm 3.0.0 is correct • rme is correct • rme.3.1 is incorrect	Specifies that scenario x will be used if all components specified by this parameter have been installed before. If no version is specified, then no specific version is required. If one version is specified, then the installer verifies that the version has been installed. Two version numbers specify the range of versions required. The value TRUE can be specified for the last scenario and indicates that the installer should use this scenario if conditions failed for previous scenarios.

ADVANCED_ CHOICE_x_TYPE	NONE
	NONEXCLUSIVE
	EXCLUSIVE
		Specifies the type of component choice.
		• NONE means there is no component choice allowed.
		• NONEXCLUSIVE allows the selection of one or more optional components.
		• EXCLUSIVE allows the selection of only one option from the list.
		
		For example:
		`ADVANCED_CHOICE_2_TYPE=NONEXCLUSIVE`
		Solaris only: If the type is NONEXCLUSIVE, the message displayed at the end of the component selection is:
		`Select one or more items using its number separated by a comma or enter q to quit.`

ADVANCED_ CHOICE_x_DEFAULT	Comma- or space-separated list	If typical or custom mode is selected, this property defines the initial selection. Each choice matches the number in *y* in the following properties. For the EXCLUSIVE scenario, this parameter should have only one choice.
ADVANCED_ CHOICE_x_y_TEXT	String, where x=1, 2, ... and y=1, 2, ...	Text shows a choice option y in scenario x.
ADVANCED_ CHOICE_x_y_TAGS	Comma- or space-separated list, where x=1, 2, ... and y=1, 2, ...	Comma- or space-separated list of component tags that are selected for installation when option y of scenario x is chosen.
ADVANCED_ CHOICE_x_y_DESCRIPTION	String (optional), where x=1, 2, ... and y=1, 2, ...	Description of components to be installed when the option y of scenario x is chosen.

Table 21-13 Table of Contents File — ADVANCED COMPONENT Sections (continued)

ADVANCED_CHOICE_x_MESG= "some message"	String	**Solaris only:** Displays the specified message if the ADVANCED_CHOICE_x_CONDITION is met. The message is displayed after the component selection menu.
ADVANCED_CHOICE_x_DESCRIPTION= "some message"	String	**Windows only:** Displays the string above the component selection. Use the "\n" (two characters) to control line breaks.

Creating the PROMPT_INSTALL_TYPE Section

The PROMPT_INSTALL_TYPE section in the table of contents file (disk.toc) specifies the installation types.

Note This section is used on Solaris only. See the "Creating the INSTALL_TYPE_SELECTION Section" section on page 21-16 for similar functionality on Windows.

Table 21-14 Table of Contents File—PROMPT_INSTALL_TYPE Section

Name	Format	Description
PROMPT_ INSTALL_TYPE_ SELECTION	Comma- or space-separated list	The available installation types: • Typical—Installs the product using the recommended settings. • Custom—Allows the user to customize the setup options.
PROMPT_ INSTALL_TYPE_ SELECTION_x	String Where x is the value of the tag PROMPT_ INSTALL_TYPE_ SELECTION	Text that describes the installation type.
PROMPT_ INSTALL_TYPE_ SELECTION_ Default	Typical Custom	The default installation type.

Example

```
[PROMPT_INSTALL_TYPE] PROMPT_INSTALL_TYPE_SELECTION=Typical,Custom
PROMPT_INSTALL_TYPE_SELECTION_Typical="Typical installation is
    recommended for all computers." PROMPT_INSTALL_TYPE_SELECTION_Custom="Custom
installation can be
    selected if you want to customize the setup options."
PROMPT_INSTALL_TYPE_SELECTION_Default=Typical
```

Creating the INSTALL_TYPE_SELECTION Section

The INSTALL_TYPE_SELECTION section specifies which installation modes (Typical or Custom) are provided.

Note This section is used on Windows only. See the "Creating the PROMPT_INSTALL_TYPE Section" section on page 21-16 for similar functionality on Solaris.

Table 21-15 Table of Contents File —INSTALL_TYPE_SELECTION Section

Name	Format	Description
OPTIONS	Comma-separated list of type names.	The list of installation modes that should be provided by this installation. For example: OPTIONS=Typical,Custom Recommended: Use Typical or Custom following the guidelines in the EDCS-207385.
DEFAULT	String. Contains one of the names listed in the OPTIONS list.	The mode that should be preselected by default. Recommended default: Typical.
DESCRIPTION	String.	The description displayed above the radio boxes in the type selection dialog. Use "\n" (two characters) to control line breaks.
<type>	String. Name should match the names in the OPTIONS list.	The text to be used as a description of the setup types. For example: Typical=Use this option in most cases Custom=Installation for advanced users

Example

```
[INSTALL_TYPE_SELECTION]
OPTIONS=Typical, Custom
Typical=Typical installation. Allows you to select components. Recommended for most users.
Custom=Custom installation. Allows you to select components and customize settings. Recommended only for
advanced users.
DEFAULT=Typical
DESCRIPTION=Choose the type of Setup you prefer, then click Next.\nThis CD includes the following
components:\nCommon Services 2.2, CiscoView 5.5, Integration Utility 1.5.
```

Note Line breaks are not allowed. For example, the entire text "`Typical=Typical installation. Allows you to select components. Recommended for most users.`" must be specified on a single line.

Understanding the BUILD Section

The BUILD section in the table of contents file (disk.toc) contains two tags, and is generated at packaging time.

Note This section is automatically generated. The developer is not required to add anything to the disk.toc file. *Do not specify this section manually.*

Table 21-16 Table of Contents File—BUILD Section

Name	Description
BUILD_ID	The build ID of the CWCS build.
ITOOLS_BUILD_ID	The build ID of the itools build.

Example

```
[BUILD]
BUILD_ID=SOL_CDONE2_2_20030115_0756
ITOOLS_BUILD_ID=SOL_ITOOLS2_2_20030115_0017
```

Creating the REQUIRED section

The REQUIRED section in the table of contents file (disk.toc) contains the list of components that must be installed before running this installation. The installer checks for these components *before* asking the user to answer other installation-specific questions. Package dependencies specified in the package properties file, on the other hand, are verified much later, after the user has finished all installation user inputs, and after running all preinstall logic.

Table 21-17 Table of Contents File—REQUIRED Section

Name	Format	Description
REQ_xxx, where xxx must match the tag and version of the component.	String	The installer expects the name to contain the tag and version (or version range) separated by the colon sign (:). For example: • REQ_cdone:2.2 verifies the presence of the cdone component, version 2.2. • REQ_dmgt:2.0.0-2.99.0 verifies that component dmgtd is installed, and its version is in the range between 2.0 and 2.99. The value should contain the message to be displayed to the user if the required component is not installed. For example, for the installer to verify if CWCS 2.2 is already installed, specify the following in disk.toc: `[REQUIRED]` `REQ_cdone:2.2=Please install CiscoWorks Common Services 3.0 before running this installation` The value must be specified on a single line.

How the Installer Processes Properties

Build image tools convert pkgpr files into newly-named platform-specific files (*.info on Windows and *.pkginfo on Solaris). The following steps describe how the installer processes package properties:

1. Initially properties are specified in *.info or pkginfo files (see the "Specifying Properties" section on page 21-19).
2. The installer overrides component properties by those specified in the *tag* section of the Table of Contents.
3. The installer verifies the presence of the ADVANCED_CHOICE_1 section. The installer verifies all conditions in ADVANCED_CHOICE_1_CONDITION. If all conditions are matched, then scenario 1 determines options in the custom installation and default components. If ADVANCED_CHOICE_1_CONDITION is not valid, then the installer tries scenarios 2, 3, and so on until the condition is valid.

4. If none of the ADVANCED_CHOICE_x_CONDITION conditions were met or there were no ADVANCED_CHOICE_x sections in the table of contents, then the component choice and default components are controlled by the sets properties VISIBLE, DEFAULT, and CHOICE properties in the COMPONENTS section of the table of contents.
5. The VISIBLE property lists the components which are displayed in the confirmation dialog if selected by you or by default.
6. The UNINSTALLABLE property in the COMPONENTS section specifies the list of components that will be listed for uninstallation.

Specifying Properties

Properties for packages and installable units are specified by developers and included in the protopackages. NMTG does not recommended specifying VISIBLE, DEFAULT, CHOICE, or UNINSTALLABLE properties for packages or installable units. For suites, these properties can be specified in *tag*.info files if the suite is planned for more than one release.

For each CD, the developer must create a table of contents. This file should be included in a protopackage, just like all other files. To be able to reuse protopackages in multiple images, you should include disk.toc in a separate protopackage, such as <tag>.cd.tar, where <tag> matches one of the packages included in the image.

This CD protopackage shall be specified as an input to the buildImage command, along with the protopackage containing the installation files and application protopackages.

Preparing Installation Protopackages

A protopackage contains one directory named after the tag. This directory contains the following subdirectories:

- runtime

  This directory contains the directories and files to be installed on the destination server. All these files will be moved during the installation to directory specified by the user.

  - On Solaris, the link /opt/CSCOpx will be set to the directory specified by the user and all files and directories under *tag*/runtime will be copied into /opt/CSCOpx.

    The runtime tree on the destination server will be a result of merging of runtime subtrees of all protopackages.

  - On Windows, if the user chooses the directory C:\Program Files\My Directory, then file *tag*\runtime\objects\myAppObjects\foo will be copied to C:\Program Files\My Directory\objects\myAppObjects\foo.

    All files from all runtime directories will be included into data1.cab. Each protopackage makes a file set. The runtime directory can be empty.

- install

  This directory contains the following files:

  - *tag*.pkgpr. This mandatory file contains package properties. On Windows, the value of the PKG property must be equal to *tag*.
  - *tag*.rul. This Windows-only optional file is the source code of installation hooks on Windows.

  - *tag*.bprops (optional). This file contains build properties each line being in the form name=value. Three values from this file are used to identify the build: PROP_ID, PROP_WHEN, and PROP_TIMESTAMP. These properties allow you to identify the build from the installed product.
  - preinstall, postinstall, preremove, and postremove. These Solaris only, optional files are installation scripts.
- disk1. All files from this directory will be copied over to the root of the CD image. The disk.toc file, or table of contents, is delivered this way. Refer to the "Creating the Table of Contents" section on page 21-11 for more details.
- cd. All files and subdirectories from this directory will be copied to the root of the CD image. Can be used for additional files required by components that do not necessarily relate to the installation framework.
- isupport. The Windows-only files from this directory will be compressed into the file group Language Independent OS Independent Files of the _user1.cab file. During installation, all these files are decompressed into the SUPPORTDIR and can be used by installation scripts.

Related Topics

See the "Including Files in the Protopackage" section on page 21-20.

Including Files in the Protopackage

Include this file in each protopackage:

- Package properties file (pkgpr)—contains name value pairs for the platforms your package supports. Specifies properties for each platform. See the "Specifying Package Properties" section on page 21-6 for more information about this file.

The following files are optional:

- Build process file (bprops)—normally generated by the build process. Specifies when and how this particular package was built. It contains build properties lines in the form of name=value. Three values from this file are used to identify the build: PROP_ID, PROP_WHEN, and PROP_TIMESTAMP. The values of the PROP_ID and PROP_WHEN are concatenated to create the value of BUILD_ID package property. The value of the PROP_TIMESTAMP is assigned to the BUILD_TIMESTAMP package property. These properties allow you to identify the build from the installed product.
- On Windows platforms, *pkg*.rul, is an optional file, which contain Windows scripts for installation. See the "Using the pkg.rul Installation File" section on page 21-34 for more information.
- On Solaris platforms, preinstall, postinstall, prerequisite, preremove, and postremove files are installation scripts for UNIX and are described later in this chapter.

All files mentioned above should be included in protopackages located in the install subdirectory and cannot be changed.

Related Topics

See the "Preparing Installation Protopackages" section on page 21-19.

Using the Installation Framework

The following topics describe the general tasks involved when using the installation framework:

- Understanding the Common Services Upgrade
- Understanding and Implementing the casuser
- Providing Licensing Information During Installation
- Installing Database Upgrades
- Handling Patches
- Application Registration with ACS during Install

Understanding the Common Services Upgrade

The CWCS team has updated the CD One 4th Edition installation and subsequent releases to disable all applications (with the exception of CiscoView) and change the ownership of the application files to the new user, casuser.

CWCS installation disables applications by unregistering daemons that do not belong to CWCS. Corresponding Windows Services will be removed. After disabling and changing file ownership, the newer version of Common Services is installed, using the installation framework upgrade procedure. As newer versions of the applications are installed, they will be able to access their data from previous versions and upgrade using the current installation framework procedure.

An additional step is required for developers updating existing code dependent on CMF 1.2 or CMF 2.2. For details on build and packaging tasks under these conditions, review the application-specific build requirements in the section “Implementing the Bin Replacment User”, which appears on page 8-36 of ENG-71441, *Bin:Bin Security Implementation Design Specification*.

Understanding and Implementing the casuser

If you have existing applications that are dependent on CMF 1.2 or earlier and must upgrade, read this topic to understand how the CWCS team has implemented security changes for owner and group.

The CWCS team has updated the CD One installation to disable all applications (with the exception of CiscoView) and change the ownership of the application files to the new user, casuser. Application developers must modify the post-install hooks andscripts to add chown and chgrp commands for each file that existed in previous CWCS or CMF releases and were not initially a part of the installation packages. This includes dynamically created files, such as data files and properties files.

To assist this process, the resetcasuser script in NMSROOT/setup/support was enhanced to allow you to randomly generate the casuser password, or enter it manually.

Related Topics

See the:

- “Understanding the Common Services Upgrade” section on page 21-21.
- “Setting Ownership for Package Files on Solaris” section on page 21-86.

Providing Licensing Information During Installation

CWCS 3.0 introduced Flex LM-based licensing as describ ed in Chapter 34, "Using the Licensing APIs."

To provide Flex LM-based licensing information during installation:

Step 1 On Windows and Solaris, add the following entry to disk.toc:

`PRODUCT_INFO=`*`Product Version`*

Where:

- *`Product`* is the short name of the product (e.g., `cm` for Campus Manager)
- *`Version`* is the product version number (e.g., `4.0`).

Step 2 On Windows only, add the same entry under the RELEASE section in disk.toc.

Installing Database Upgrades

This topic provides information about the structure and APIs needed to upgrade the database. This feature relies on the implementation of the dbupdate.pl script script provided by CWCS. This feature is not required. Use this database upgrade information only if your team is using or integrating with the CWCS Server database.

NMTG developers should refer to ENG-29984 for requirements if you are bundling with RME CD (copackaging).

The following topics are covered:

- Upgrade Installation Paths and Strategies
- About the CWCS Upgrade Mechanism
- Adding Unauthenticated URLs
- Overriding the Dependency Handler

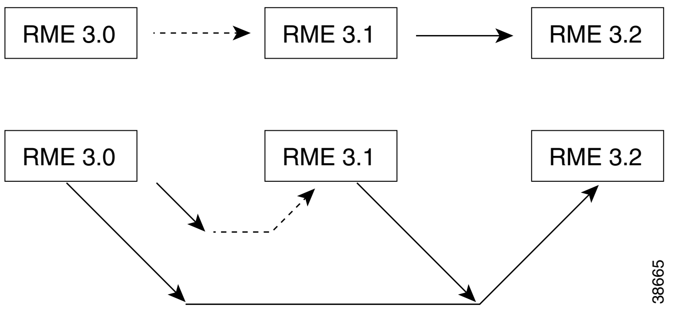
Upgrade Installation Paths and Strategies

There are two main approaches to upgrading the database.

- An incremental approach implements a set of steps, each of them making adjustments to bring the database to the next version. These steps are executed as needed, depending on the version of a product already installed. You can implement each step as a separate script. This enables the required additional steps for each new version and only takes care of changing the database from the latest previous version to the current version.
- A one-step approach that brings the database to the current version from any previous state.You can convert all previous versions of the database to the current one.

Figure 21-1 depicts both approaches for RME. Each arrow represents an upgrade script. The solid arrows represent scripts that have been implemented to upgrade Rigel releases. The dashed arrows represent scripts that will handle an upgrade to RME3.1. The dotted arrows represent scripts to handle upgrade to RME 3.2 sometime in the future.

Figure 21-1 RME Upgrade Approaches



The incremental approach assumes that the script upgrading RME3.0 to RME 3.1 handles only the upgrade from 3.0.

With the one-step approach, the script that you used to upgrade 3.0 to 3.1 needs to be replaced by the new script that is capable of upgrading from both 3.1 and 3.2. The complexity of such a script increases for each release, because the script must handle upgrades from all previous versions.

About the CWCS Upgrade Mechanism

CWCS has one mechanism to modify the database. The *dbupdate* script is integrated into the installation and enables you to register your database upgrade scripts in preinstall (Windows) or prerequisite (UNIX) hooks. Registered scripts are executed at the end of installation after all runtime files have been delivered. The database is upgraded in place.

The upgrade mechanisms described in this topic include:

- Registration API for Upgrades
- API for Package Upgrade—CopyOut API
- Uninstall Before Upgrade
- Function Prototypes

Registration API for Upgrades

The functions described here can be used in preinstall (Windows) or prerequisite (UNIX) to register the database upgrade scripts for dbupdate. The following table describes the functions and recommended usage.

Function	Recommended usage
RegisterDBScript	Use for drop-ins
RegisterDBScriptByVersion	Applications that use incremental approach
RegisterDBScriptByVersionEx	Applications that use a one-step approach

API for Package Upgrade—CopyOut API

To provide a useful package-based CopyOut feature, a CopyOut API has been implemented. This saves the package developers time because they do not have to use just the hook to launch the copyout script. The reason is that the "just the hook" approach in the installation framework cannot verify the total disk space required for all copyouts, because copyouts are package-independent.

You may also have to do some housekeeping work. With CopyOut API solution, multiple manifest files, different storage destinations for each package, attributes for query such as DATE_COPYOUT, PREVIOUS_VERSION, and so on can be supported. Most importantly, disk space verification is possible.

You can call this API inside the prerequisite (preinstall on Windows in pkg.rul) script. The actual copyout does not happen until all prerequisites are satisfied. The CopyOut API stores all parameters into a temporary file.

At the end of all prerequisite verification, the API verifies copyout storage requirements, plus disk space required for the installation if copyout destination happens to be in the same partition of *NMSROOT*. After the copy is done, the manifest file is copied to the storage destination location. Pkgname.properties are also created and copied to the storage destination location containing name-value pairs:

```
COPYOUT_DATE=YYYYMMDDHHMM
SOURCE_PATH=Path to copy data from specified in IF_CopyOut call
OVERWRITE_MODE=Y|N specified in IF_CopyOut call
PREV_VERSION=prevmajor.prevminor
PREV_PATCHVER=prevpatch
NEW_VERSION=newmajor.newminor
NEW_PATCHVER=newpatch
```

Pkgname.info contains an entry pointing to this pkgname.properties. Therefore multiple entries of pkgname.properties may exist if CopyOut is executed more than once for each package using different storage locations. The property is named COPYOUT_XYZ starting with XYZ=001 and up.

For example:

```
COPYOUT_001=/opt/CiscoWorksOldData/conf/CSCOani.properties
COPYOUT_002=/opt/CiscoWorksOldData/etc/CSCOani.properties
```

Syntax	`prototype IF_CopyOut(SourceDataPath, ManifestFileName, DestPath, OverwriteFileMode);`		
Description	Instructs the installation framework to collect all parameters for future verification and actual copyout.		
Parameters	**Field**	**Type**	**Description**
	SourceDataPath	String (input)	Source path to get root data specified in manifest file. For example, *$NMSROOT*/conf
	ManifestFileName	String (input)	File that contains the list of files to be copied out relative to SourceDataPath (ani.mfst). Files are in the same directory of the setup script (for example, disk1 location)
	DestPath	String (input)	Where to copy out. For example, /opt/CW2000/CiscoWorksOldData
	OverwriteFileMode	String (input)	Designates whether to overwrite existing file during the copy using "Y"/"N".

Examples

Solaris prerequisite script:

```
#!/bin/sh

. $SETUPDIR/commonscript.sh
. $SETUPDIR/setup-lib.sh
# checking for copyout condition
# if this is an upgrade, call IF_CopyOut
    if [$NEED_COPY_OUT eq 1]; then
        IF_CopyOut "$NMSROOT/conf" "ani.mfst"
            "/opt/CW2000/CiscoWorksOldData" "Y"
```

The same script on Windows:

```
 Function ani_preinstall()
    NUMBER nvType, nvSize, nRc, nNeedCopyout;
    STRING bInstallingFirstTime;
Begin
    // checking for requirements
    nNeedCopyOut = 0;
    // check for upgrade case to call IF_CopyOut
    if (nNeedCopyOut = 1) then
        IF_CopyOut((NMSROOT^"conf", "ani.mfst",
            NMSROOT^"OldData", "Y");
    endif;
end;
```

Uninstall Before Upgrade

The installation framework provides a hook to trigger package removal before the package is installed. For the package which prefers a clean installation, you can specify a combination of UNINSTALL=UPGRADE|PATCH|REINSTALL in the pkg.info(pkg.pkgpr) to make this work.

Caution Data files belonging to the existing package will be deleted during the package removal. Performance is also affected. Remove the package only if there are major changes for the package.

Uninstallation for package downgrade is not supported. The action taken depends on the combination of the flags given in the UNINSTALL:

- UPGRADE—only uninstall the package if this is a package upgrade (CD image major.minor or is newer than the existing one. The patch version is ignored).

  For example, UNINSTALL=UPGRADE uninstalls package if package is upgraded, but not during reinstall or patching.

- PATCH—only uninstall package if CD image major.minor version of the package is the same as existing one, and the CD image patch version is newer than the existing patch version.

  For example, UNINSTALL=UPGRADE PATCH uninstalls package if it is upgraded or patched, but not during reinstall.

- REINSTALL—only uninstall if the CD image major.minor and patch version are the same as the existing one. Note that this only makes some difference if the package provides additional cleanup in postremove (SOL) or uninstall (Windows).

  For example, UNINSTALL=UPGRADE REINSTALL uninstalls the package if the package is upgraded or reinstalled, but not during patch case.

Function Prototypes

```
RegisterDBScript(<dsn>, <script>, <args>);
RegisterDBScriptByVersion(<fromVersion>,<dsn>, <script>, <args>, <reserved>);
RegisterDBScriptByVersionEx(<fromVersion>,<dsn>, <script>, <args>, <reserved>);
```

Where:

- *dsn* is the DSN of the database;
- *script* is full installer of the script;
- *args* is the string of arguments to be passed to the script;
- *reserved* This parameter should always be empty.

Windows Example

On Windows the infsm.rul file contains:

```
function infsm_preinstall()
STRING argString, scriptName;
begin
    argString = “dsn=rme,” + NMSROOT ^
        “IDS\\inventory\\desfiles\\fastswitch_wmod,CSCOinfsm”;
    WriteLogFile(“preinstall for infsm package”);
    scriptName = NMSROOT ^ “IDS\\inventory\\scripts\\main.pl”;
    RegisterDBScriptEx(“rme”, scriptName, argString, ““);
end;
```

Solaris Example

On Solaris, the infsm.rul file contains:

```
. /opt/CSCOpx/etc/commonscript.sh
RunRequestScript CSCOinfsm
#main
echo “preinstall for CSCOinfsm”;
argString=”dsn=rme,/opt/CSCOpx/IDS/inventory/desfiles/fastswitch_wmod,CSCOinfsm”;
scrName=”/opt/CSCOpx/IDS/inventory/scripts/main.pl”;
RegisterDBScriptEx “rme” $scrName $argString “ ";
```

Adding Unauthenticated URLs

This API allows applications to use URLs without authentication.

Applications need some of their URL calls to be allowed without any authentication. If the system checks for authentication for every URL, some functions of the application may fail.

This API is part of the CWCS installation framework, and provides an option to use application URLs without authentication. It uses a file, allow_files.conf, that contains a list of URLs that should be allowed without authentication.

To make use of this API, applications should:

- Identify the list of relative URL calls to be allowed without authentication. You can find this by checking error_log for any HTTP forbidden errors.
- Create a file, addURLs.txt, in disk1, and include all such URL calls (that is, the URL calls to be allowed without authentication).

  Example of addURLs.txt:

  ```
  /CSCOnm/servlet/com.cisco.nm.cmf.servlet.webreg.csNavServlet
  /help/cmf/jplug_enable.html
  ```

```
/JSP/cmf/security/AutoLogin.jsp
```

You can specify relative directory paths (with respect to web server) also in addURLs.txt. For directories, the format is as follows:

```
DIR=relative dir
```

For example, to allow the /opt/CSCOpx/htdocs/swintemp directory without the authentication check, you must add the following in addURLs.txt:

```
DIR=/swimtemp
```

`During the installation of applications, the installation framework appends the contents of addURLs.txt to allow_files.conf.`

Overriding the Dependency Handler

This feature allows you to bypass the dependency handler during installation. Use it when normal policies are believed to be inappropriate (for example, to rollback a patch).

Caution This tool is meant to help developers get out of trouble, and is only recommended for specific situations.

Refer to the engineering spec, ENG-29971, for details about using the override.

Handling Patches

This topic provides details about how to implement patching support and supplies a checklist for developers. For examples on how to patch a CD, refer to the "Example: Making a Patch CD" section on page 21-28.

This topic covers:

- Patch Policy
- Creating a Patch

Patch Policy

Installation provides a way to prepare point patches as well as patch releases. Rollback is not supported. A typical patch contains new files that can be added to a previously installed product. *Patches must be cumulative.*

Creating a Patch

When creating a patch keep in mind that the version of a package is defined by the VERSION and PATCHVER package properties (see the "Specifying Package Properties" section on page 21-6). Developers must change the *pkg*.pkgpr file. Normally, the PATCHVER value should be increased for each patch.

The installer must know whether the package is a full or a sparse package:

- A **full** package contains all the runtime files that belong to the package.
- A **sparse** package contains only the files that are modified by the patch. A sparse package is indicated by adding the following property to the <*tag*>.pkgpr file:

```
SPARSE=YES
```

Patch Mode Installation

Patch mode enables patch delivery of one or more packages provided that a full release had been installed before. Patch mode turns off the standard dependency verification. Each package available on CD will be installed only if the same or older version of that package had previously been installed.

Packages available on the CD are skipped if the newer version of the same package has been installed or this package has not been installed. There will be no error message in this case.

The patch mode installation is enabled by using the PATCH_TAGS property in the table of content. See the "Example: Making a Patch CD" section on page 21-28 for sample code to create a CWCS patch CD.

To create a patch:

Step 1 Create the table of contents for patch or patch release.

a. Take the disk.toc of the previous release and make a copy of it.

b. Modify the NAME and VERSTR to show the name and version of a patch release.

c. Adjust the CHOICE, VISIBLE and DEFAULT properties to control the user interface.

The TAGS and UNINSTALLABLE properties remain unchanged if the patch release has installable units with all packages.

If release delivers only selected packages and does not have any new package, the patch mode should be used. In this case the UNINSTALLABLE property is removed and the TAG property is replaced by the PATCH_TAGS property. The PATCH_TAGS property contains the list of packages included into release.

Step 2 Fix the project file.

Note This step is required only when the NMTG build environment is used.

NMTG recommends that you create a full version on a regular basis. You can add only one CD image in the project file at this stage. Later, you can add more than one image created by one project, as needed.

Step 3 Modify sources.

Step 4 Roll up the appropriate VERSION and PATCHVER.

Note The same uninstallation hook is responsible for removing the package both before and after applying a patch. Uninstallation of a patch is not supported. Therefore you can uninstall a package before or after a patch, but the patch cannot be reverted.

Step 5 If the package is sparse (contains only the files that are modified by the patch), add the following line to the <*tag*>.pkgpr file:

```
SPARSE=YES
```

Example: Making a Patch CD

The installation framework allows you to create a CD with patches. A patch CD has selected packages that can only be installed on top of the entire product. In the Patch mode, installation does not check dependencies, instead it verifies that an older version of the same package had been installed before. If an older version of a package has not been installed or the newer version of a package has been installed, that package is skipped without error messages. For instructions about making patches, see the "Handling Patches" section on page 21-27.

This section provides an example of making a patch for CWCS. The first step describes creating a CD with the patch for a package. Next step describes how to make a CD containing patches for two related packages. This section uses my_app, my_appdev, and similar text to illustrate any network management package.

The following topics are covered:

- Patching my_appdev
- Patching my_appdev and my_app

Patching my_appdev

Perform the following steps to create your my_appdev patch.

1. Updating the *.pkgpr
2. Building the New Protopackage
3. Making the Table of Contents File
4. Creating the CD

Updating the *.pkgpr

Update your package by adding the patch version name value pair to your pkgpr file.

For example, if your CMF 1.0 package had version 1.0 and the my_app.pkgpr had the following properties:

```
NT:AIX:HPUX:SOL:
DESC=My Network Management Application
NAME=My Application
VERSION=3.0
…
```

Update your pkgpr file to include the PATCHVER:

```
NT:AIX:HPUX:SOL:
DESC=My Network Management Application
NAME=My Application
VERSION=3.0
PATCHVER=1
…
```

Building the New Protopackage

For information on protopackages, see the "Step 3: Prepare the Make Image on Solaris" section on page 21-103.

Making the Table of Contents File

The following is an example of the table of contents file (see the "Creating the Table of Contents" section on page 21-11).

```
[RELEASE]
NAME=CWCS3.0 Patch CD Version 1
VERSTR=automatic build
REGISTRY_ROOT=SOFTWARE\Cisco\Resource Manager\CurrentVersion

[COMPONENTS]
PATCH_TAGS=my_appdev
```

```
UNINSTALLABLE=
VISIBLE=
CHOICE=
DEFAULT=ALL

[REQUIRED]
REQ_cdone=Install Common Services 3.0 first
```

A similar file has to be created for UNIX platforms if packages listed in the PATCH_TAGS property have different tags; otherwise, the same file can be used for all platforms. The table of contents file has to be built into a protopackage and must have the name *disk.toc*. Solaris also requires the *configureMe* file:

```
nm-build3-sol251% tar -tvf cmf.patchcd.tar
tar:blocksize = 11
drwxrwxr-x 8186/25       0 Dec 1 14:55 1999 cmf/
drwxrwxr-x 8186/25       0 Dec 1 14:55 1999 cmf/runtime/
drwxrwxr-x 8186/25       0 Dec 1 14:55 1999 cmf/disk1/
-rw-r--r-- 8186/25     199 Dec 1 14:35 1999 cmf/disk1/disk.toc
-rwxr-xr-x 8186/25    1177 Nov 24 14:27 1999 cmf/disk1/configureMe
```

Note On Windows, only the *disk.toc* file is required.

The following text is an example of part of the *configureMe* file.

```
AIX_MIN_RAM=`echo "128 * 1024" | bc`; export AIX_MIN_RAM
AIX_MIN_SWAP=`echo "$AIX_MIN_RAM * 2" | bc`; export AIX_MIN_SWAP
AIX_OS_VERSION="5.5"; export AIX_OS_VERSION
DEF_IU_ROOT="/opt/CSCOpx"; export DEF_IU_ROOT
DISKSPACE=0; export DISKSPACE
HP_MIN_RAM=`echo "256 * 1024" | bc`; export HP_MIN_RAM
HP_MIN_SWAP=`echo "1024 * 1024" | bc`; export HP_MIN_SWAP
HPX_OS_VERSION="5.5"; export HPX_OS_VERSION
INSTALL_MODE="NEW"; export INSTALL_MODE
IU_DBDIR="${IU_ROOT}/db/data";export IU_DBDIR
IU_NAME="CSCOpx"; export IU_NAME
IU_ROOT="/opt/CSCOpx"; export IU_ROOT
LOG_NAME="ciscoinstall.log"; export LOG_NAME
PRODUCT="CSCO NM"; export PRODUCT
SERVER_EXP="[C]SCOpx"; export SERVER_EXP
SOL_MIN_RAM=`echo "128 * 1024" | bc`; export SOL_MIN_RAM
SOL_MIN_SWAP=`echo "$SOL_MIN_RAM * 1" | bc`; export SOL_MIN_SWAP
SOL_OS_VERSION="5.7 5.8"; export SOL_OS_VERSION
SWMODOPTS="-xloglevel=0"; export SWMODOPTS
SWOPTIONS="-x reinstall=true -x reinstall_files=true -x allow_multiple_versions=true -x
write_remote_files=true -x autoselect_dependencies=false -x match_target=false"; export
SWOPTIONS
UPGRADE_VERSIONS="2.* 3.0"; export UPGRADE_VERSIONS
YES_PUMP=${SETUPDIR}/install/yes.sol; export YES_PUMP
REQ_OS="SunOS"; export REQ_OS
```

Creating the CD

Use the following command to create the CD:

```
perl buildImage -r -d d:/cmfPatchImage path/is5.runtime.tar path/cmf.patchcd.tar
path/my_pkg.runtime.tar
```

For Solaris, use the pkgtools.runtime.tar file instead of is5.runtime.tar.

A patch is installed the same as a normal installation, by running the setup.sh script on Solaris.

Patching my_appdev and my_app

You can create a patch CD containing patches for both my_appdev and my_app using the very same processes presented in the "Patching my_appdev" section on page 21-29. The only difference is that my_app must be added to the PATCH_TAGS property in the table of contents.

The command to create the CD is (all on one line):

```
perl buildImage -r -d d:/cmfPatchImage path/is5.runtime.tar
    path/cmf.patchcd.tar path/my_appdev.runtime.tar
    path/my_app.runtime.tar
```

Application Registration with ACS during Installation

When the CiscoWorks server is configured for ACS Login mode and an application /Service-PackP/IDU/drop-in is installed, install framework attempts to register all the tasks of all applications currently installed on the server with ACS. In this process, any customization of roles done in ACS is overwritten for all applications.

Starting from Common Services 3.0 Service Pack 2 (CS3.0 SP2), Common Services install framework minimizes the risk of applications getting re-registered with ACS during install/re-install/upgrade.

For Applications based on CS 3.0 SP2/Corresponding ITOOLS or higher:

If server is in ACS mode and the particular IDU/Application/SP requires new tasks to be added, appropriate warning will be provided and that Application's tasks alone will be registered.

New tasks in addition to old tasks for that applications will be registered with ACS.

Applications need to pass the following information to the intall framework, so that registration with ACS is done accordingly:

- MDC Name
- Version
- New tasks

Note Enhancement requests have been opened against ACS. When those are addressed in ACS, the above approach will be revised.

Applications need to add a file `acsmap.txt` under their disk1 directory. This file will be used for populating the ACS registration mapping file that CS maintains for selective registration of tasks with ACS. Install framework will also use this file to decide whether a Warning has to be displayed.

Contents of `acsmap.txt` should be of the format:

MDC_name;app_name;app_version (with patchver)=<New tasks to be registered -Y/N>

Example: If RME 4.0.2 has new tasks to be registered with ACS, the contents will be:

```
rme;rme;4.0.2=Y
```

If an application has more than one MDC (to be registered with ACS), all of them should be mentioned in a separate line.

Example:

```
Rme1;rme;4.0.2=Y
Rme2;rme;4.0.2=N
Rme3;rme;4.0.2=Y
```

The application name mentioned here should preferably map to a corresponding <appname>.info file under *NMSROOT*/setup

Example:

```
cmf, rme, cvw1
```

where there are corresponding info files. If needed, a displayable "Application name" can be picked from the INFO file. [PRODNAME attribute in the <*app*>.info file will be used for this purpose].

If the file `acsmap.txt` is not available under disk1, the MDC name of the product will be assumed as the "application name" as well.

Applications not based on CS -ITOOLs can choose to use the CLI script `AcsRegCli.pl`. The relevant documentation of such applications should be updated to recommend registering from CLI, as applicable.

`AcsRegCli.pl` command line script can be used to register an application without affecting the registration status of other applications. The script is located at *NMSROOT*/bin. `AcsRegCli` can be run only when the CiscoWorks server is in ACS mode.

`AcsRegCli.pl` has the following options:

- listRegApp—list the applications registered with ACS in the current CiscoWorks server.
- listNotRegApp—list the installed applications that are not registered with ACS in the server.
- register <MDCName>—register an application with ACS. <MDCName> is the name by which an application will be registered with ACS. To know this value, run `AcsRegCli.pl` with the option -listRegApp or -listNotRegApp.
- register all— register all the installed applications with ACS.

Refer Appendix of CS 3.0 SP2 SFS - EDCS-447583, for more details

Windows Installation Reference

This topic covers the following reference information for Windows:

- Setting File Permissions During Installation on Windows
- Writing Windows Scripts
- Using the Windows Installation APIs
- Using Windows Build Tools
- Customizing the Installation Workflow for Windows

Setting File Permissions During Installation on Windows

With the release of CWCS 3.0, the Windows “Everyone” group *does not* have read permissions for files under NMSROOT. It is not possible to set permissions to individual directories and files; only administrators and casuser have permissions to NMSROOT and directories or files under it.

Writing Windows Scripts

Using the installation framework APIs provided with CWCS (see the "Using the Windows Installation APIs" section on page 21-37), you can write scripts that will allow you to specify any requirements and enforce constraints on a package on a Windows platform.

In addition to the basic scripts described in the following sections, you can write additional scripts to customize the logic flow of the installation (see the "Customizing the Installation Workflow for Windows" section on page 21-76).

The following topics describe how to write Windows scripts for your package:

- Using the Windows Installation Hooks
- Using the pkg.rul Installation File
- Using Installer Global Variables
- Preloading the Global List, lAnswerFile
- Reducing Windows Installation Time

Using the Windows Installation Hooks

You can use the following hooks to perform the required functions for Windows package installation:

Table 21-18 Windows Installation Hooks

Hook Type	Description
Preinstall	The installer executes this hook before file transfer. The code in this hook should not make any changes on the target system because the user can abort installation later. Ensure that you stop execution and get the data for the components that have been installed. This data specifies the disk space required and questions to be asked of the installer. If the information obtained by this script is required by another script, such as preinstall, then use SetPackageProperty and GetPackageProperty APIs. If preinstall wants the installation to fail, it should call the function SetAbortFlag (see the "Using the Windows Installation APIs" section on page 21-37 for details).
Postinstall	This hook is executed after file transfer. Actual configuration of component. This is the correct place to register daemons.
Uninstall	This hook is executed when the component is uninstalled right before removing files. This is a good place to unregister daemons. You can use this to clean up files and directories that were created when the product is running. For example, use this script to remove log directories and files.

Hooks are executed in the following sequence:

1. Preinstall for all components.
2. File transfer for all components.
3. Postinstall for all components.

At uninstallation time, hooks are executed in the following sequence:

1. Hooks for all components are uninstalled.
2. Files for all components removed.

It is important to understand the order in which the hooks are executed. The installer follows these rules:

- If component A depends on component B, hooks for A are executed *after* corresponding hooks for B.

- The installer leaves out hooks if it decides that a particular component should not be processed. It always leaves out a component if its pending version is lower than the installed version. It can leave out a component if it is optional and not required by any other component.
- The installer executes preinstall hooks if their pending version or patchver is the same as the installed version.
- For installable units, the preinstall hook is executed *before* hooks of packages that belong to this component; postinstall is executed *after* the hooks of packages.

At uninstallation, the installer follows these rules:

- If component A depends on component B, hooks for A are executed *before* corresponding hooks for B.
- For installable units, the uninstall hook is executed *after* the uninstall hooks of components that belong to this installable unit.
- Shared packages are not removed and their uninstall hooks are not executed unless all installable units they belong to are uninstalled.

Note In general, hooks should not rely on any order of execution. It is especially important to provide the same hooks for both major releases and for upgrades.

Using the *pkg*.rul Installation File

InstallShield has its own scripting language. If you use the *.rul file, the hooks are named specifically for the InstallShield. If all you plan to do with your package is to drop it in the runtime tree, you do not need to reconfigure the installation, and the *.rul file is not required. The *pkg*.rul file contains Windows scripts for installation. This is an optional file that contains InstallShield5 code for a hooks for a package or installable unit. The *pkg* must match the name of the pkgpr file described above as well as the value of PKG property.

The *pkg*.rul file begins with the line:

```
declare
```

followed by #define statements and function prototypes. Prototypes are needed only for additional function, not for hooks themselves. For example:

```
declare
#define MY_CONSTANT5
#define ANOTHER_CONSTANT “name of file”
prototype MyFunction(STRING, BOOL, STRING);
```

After that bodies for hook functions and additional functions are included. Hook functions are InstallShield5 functions with special names.

pkg_<hook_type>,

where *pkg* is package tag and *hook_type* is one of preinstall, postinstall, or uninstall hooks. Prototypes for hook functions are generated automatically and cannot be included into the *pkg*.rul file.

For example:

```
function mypkg_preinstall()
begin
    StopService(“srv”);
end;

function mypkg_postinstall()
    NUMBER nRc;
```

```
begin
    DmgrRegister("mydaemon", ….);
    nRc = MyFunction("mydaemon", TRUE, "parameter");
end;

function mypkg_uninstall()
begin
    DmgrUnregister("mydaemon", …);
end;

function MyFunction(dmName, flag, msg)
begin
    …
end;
```

Functions can take advantage of APIs and global variables. See the "Using Installer Global Variables" section on page 21-35 for more details.

You can call the Core Client Registry from an installation .rul file using ccraccess.dll. See the "Using the CCRAccess DLL" section on page 13-50.

Using Installer Global Variables

Caution If you want to create packages to be used with CWCS, *do not change* these global variable values.

The following global variables are provided by the installer and can be used directly in installation hooks. Future marketing requirements may change these values.

Variable Name	Description
TOC_PRODUCT_NAME	The full product name (CWCS). Use this name in user messages.
TOC_SHORT_PRODUCT_NAME	The short product name (also CWCS). Use this name internally to prefix other names (for example, CWCS Daemon Manager).
TOC_REGISTRY_ROOT	The main registry key for the product is (HKEY_LOCAL_MACHINE\SOFTWARE\Cisco\Resource Manager\CurrentVersion).
TOC_VENDOR	The vendor name (Cisco Systems).

To override these values, add a line to the RELEASE section of the table of contents without the TOC_ prefix. For example, the following overrides the TOC_REGISTRY_ROOT:

```
[RELEASE]
REGISTRY_ROOT=HKEY_LOCAL_MACHINE\....
```

Preloading the Global List, lAnswerFile

The installation framework automatically preloads the answer file into the global list lAnswerFile. The answer file is an ASCII file that provides the required inputs for quiet installations.

Note Quiet mode is usually used internally; customers should run the installation without specifying an answer file. However, quiet mode is important for test automation and should be fully supported.

The answer file is a plain ASCII file consisting of a name=value pair on each line:

```
#--- begin answer file
#--- hash sign (#) is allowed to mark comments
adminPassword=admin
destination=d:\cscopx
systemIdentityAccountPassword=admin
#--- end of answer file
```

The CWCS installation processes the properties described in Table 21-19. If your application image includes CWCS, make sure that the mandatory properties are included in the answer file. If your application image does not include CWCS, you can ignore these properties.

Assuming the above file was named `c:\answerfile`, you could call it as part of a silent installation as follows:

```
setup.exe QUIET answerfile=c:\answerfile.
```

Table 21-19 Answer File Properties

Property	Description
casuserPassword	If casuser does not exist by the time of installation, the framework generates random password for casuser. • If the random password is successful, then no input is required. • If the random password fails, installation opens a dialog requesting new password. In quiet mode, installation attempts to load the casuserPassword from the answer file. If no casuserpassword is specified in the answer file, installation attempts random password, and might fail if the random password does not pass the company policy.
destination	Allows quiet installation to install into directory other than c:\Program Files\CSCOpx. It is optional; if not specified, the installation is installed in c:\Program Files\CSCOpx.
adminPassword	Specifies the login password for the admin user, and is MANDATORY. It is only used during an original installation; reinstallation in quiet mode does not change the password.
systemIdentityAccountPassword	Password for the System Identity Account. This is mandatory.

Related Topics

For the similar procedure on Solaris, see the "Creating the Answer File" section on page 21-89.

Reducing Windows Installation Time

Developers trying to reduce total Windows installation time will want to avoid rewriting preinstall and postinstall scripts. Apart from requiring much additional developer time, this is error prone.

However, you can save substantial installation time by:

1. Editing your scripts to eliminate any global function- and variable-name conflicts.

2. Once your scripts are free of these naming conflicts: Combine the existing pre- and post-installs into a single script, which can be processed automatically.

The existing build tools will attempt to do this by default (and will fail to do so if there are name conflicts), but developers have full control over whether scripts are combined or not, and which scripts are combined. To combine scripts under your control, add to disk.toc a SCRIPTS entry of the following form:

```
[SCRIPTS] combinedScript=[all_]sourceScript,…
```

Where:

- `combinedScript` is the name of the resulting
- `all_` designates that all children of the component whose script is specified in `sourceScript` need to be put into single scripts.
- `sourceScript` is the script whose children you want to combine. To specify multiple scripts, separate them with commas.

For example, to combine all CWCS scripts:

```
[SCRIPTS] cdone_a=all_cdone,all_cmfj2
```

On Windows platforms, you can also save installation time by replacing any calls to CCRAccess.exe with calls to CCRAccess.dll. See the "Using the CCRAccess DLL" section on page 13-50.

Using the Windows Installation APIs

This topic covers the APIs you can use in package-specific hooks (install shell scripts for Windows). The functions you can perform using these APIs include:

- Accessing and Setting Package Properties to Perform Version Comparisons
- Controlling Responses to Terminated Installations
- Processing Name=Value Pairs
- Sending Informational Messages to a Log File
- Informing the Installer That a Component Requires More Space
- Registering and Unregistering CWCS Daemons
- Running Commands in a Shell
- Locating the Root Directory Path Name
- Registering and Controlling Windows Services
- Using Generic Utilities
- Managing Passwords
- Configuring Tomcat
- Controlling Reboots

Accessing and Setting Package Properties to Perform Version Comparisons

Use these APIs to access and set package properties and to perform different version comparisons:

- CompareVersion
- CompareVersionTo

- GetInstalledPackageVersion
- GetPackageProperty
- IsInstalled
- LoadPackageProperties
- PROP_DIR
- RecordKeyValue
- SavePackageProperties
- SetPackageProperty
- VersionToMajorMinor
- VersionToMajorMinorPatch
- IsVersionInRange
- isPackagePending

CompareVersion

```
prototype CompareVersion(STRING);
prototype CompareVersionEx(STRING, BYREF NUMBER, BYREF NUMBER,
    BYREF NUMBER);
```

Compares the version of the component being installed with the one installed before. CompareVersionEx also returns the version and patch level of the latest.

Arguments

Field	Type	Description
1	STRING (input)	Package/installable unit/suite tag. Contains information about package version.
2	NUMBER (output)	Major package version number.
3	NUMBER (output)	Minor package version number.
4	NUMBER (output)	Patch level version number.

Return Values

Value	Description
0	Versions are the same
<0	Pending is lower, downgrade.
>0	Upgrade.
1	Version is the same, patch level of pending component is higher.
2	Major version is the same, minor version of pending component is higher.
3	Major version of pending component is higher or it is being installed for the first time.
-1	Version is the same, patch level of pending component is lower.
-2	Major version is the same, minor version of pending component is lower.

Value	Description
-3	Major version of pending component is lower.
-99	Pending version not found.

CompareVersionTo

```
prototype CompareVersionTo (STRING, NUMBER, NUMBER, NUMBER, NUMBER);
```

Compares pending or installed version of component to given values.

Arguments

Field	Type	Description
1	STRING (input)	Package/installable unit/suite tag. Contains information about package version.
2	NUMBER (input)	Installed/pending flag: • RM_PKGPROP_PENDING—Package installed or package on CD. • RM_PKGPROP_INSTALLED—Previous package installed. • RM_PKGPROP_ANY—Package with latest installation version. • RM_PKGPROP_CD—Package on CD.
3	NUMBER (input)	Major package version number.
4	NUMBER (input)	Minor package version number.
5	NUMBER (input)	Patch level version number.

Return Values

Value	Description
0	The same.
<0	Installed/pending/latest is lower.
>0	Installed/pending/latest is higher.
1	Version is the same, patch level of installed/pending/latest component is higher.
2	Major version is the same, minor version of installed/pending/latest component is higher.
3	Major version of installed/pending/latest component is higher.
-1	Version is the same, patch level of installed/pending/latest component is lower.
-2	Major version is the same, minor version of installed/pending/latest component is lower.
-3	Major version of installed/pending/latest component is lower.
-99	Pending version not found.

GetInstalledPackageVersion

```
prototype GetInstalledPackageVersion (STRING, BYREF STRING, BYREF NUMBER);
prototype GetPendingPackageVersion (STRING, BYREF STRING, BYREF NUMBER);
```

Determines the version and patch version of component that had been installed before this installation (GetInstalledPackageVersion) or is being installed (GetPendingPackageVersion).

Arguments

Field	Type	Description
1	STRING (input)	Package/installable unit/suite tag. Contains information about package version.
2	STRING (output)	Major version and minor package version number.
3	NUMBER (output)	Patch level version number.

Return Values

Value	Description
1	Component is found.
0	Component is not found.

GetPackageProperty

```
prototype GetPackageProperty(STRING, STRING, BYREF STRING, NUMBER);
```
Get the value of package property.

Arguments

Field	Type	Description
1	STRING (input)	Package/installable unit/suite tag. Contains information about package version.
2	STRING (input)	Name of the property to be retrieved.
3	STRING (output)	Property value.
4	NUMBER (output)	Installed/pending flag: • RM_PKGPROP_PENDING—Package installed or package on CD. • RM_PKGPROP_INSTALLED—Previous package installed. • RM_PKGPROP_ANY—Package with latest installation version. • RM_PKGPROP_CD—Package on CD.

Return Values

Value	Description
0	Property extracted successfully.
-1	Property not extracted.

IsInstalled

```
prototype IsInstalled(STRING);
prototype IsInstalledEx(STRING, BYREF STRING, BYREF STRING);
```

Verifies if package, installable unit, or suite has been installed before this installation started. The IsInstalledEx function also returns version and information string if the component is found.

Arguments

Field	Type	Description
1	STRING (input)	Package/installable unit/suite tag. Contains information about package version.
2	STRING (output)	Major and minor version number. Uses the format major.minor.
3	STRING (output)	Component information as version *major.minor* installed *date*

Return Values

Value	Description
0	Component is found.
-1	Component is not found.

LoadPackageProperties

```
prototype LoadPackageProperties(STRING, LIST, NUMBER);
```

Loads property file into string list.

Arguments

Field	Type	Description
1	STRING (input)	Package/installable unit/suite tag. Contains information about package version.
2	LIST (output)	Valid string list. Functions described in Properties API can be used to obtain or modify individual properties from this list.
3	NUMBER (input)	Installed/pending flag. Start looking from pending. If not found, try the one that is already installed. • RM_PKGPROP_PENDING—Package installed or package on CD. • RM_PKGPROP_INSTALLED—Previous package installed. • RM_PKGPROP_ANY—Package with latest installation version. • RM_PKGPROP_CD—Package on CD.

Return Values

Value	Description
0	Loaded successfully.

PROP_DIR

```
STRING PROP_DIR;(Global variable)
```

Use this global directory name for temporary files. This directory is automatically removed after installation is completed. To avoid file name collisions do not use extension .info or .temp.

RecordKeyValue

```
prototype RecordKeyValue(STRING, STRING, STRING, STRING, BOOL);
prototype RestoreKeyValue(STRING, STRING, STRING, STRING, BOOL);
```

Save and restores Windows registry key values. Keys are saved permanently, and could be saved at installation time and restored at uninstallation. Each package has its own namespace to save or restore registry key values.

Arguments

Field	Type	Description
1	STRING (input)	Package tag.
2	STRING (input)	Key name.
3	STRING (input)	Key value name.
4	STRING (input)	New key value name.
5	BOOLEAN (input)	Save mode: • For RecordKeyValue, set to TRUE to override the saved value. If last parameter is FALSE and saved value is found, value is preserved. • For RestoreKey Value, controls situation when saved value is not available. If TRUE, the value specified by key name and key value name is removed.

Return Values

None

SavePackageProperties

```
prototype SaMvePackageProperties(LIST);
```

Saves preloaded properties of current component from string list.

Arguments

Field	Type	Description
1	LIST (input)	Valid string list initialized by the LoadPackage Properties function.

Return Values

None

Notes

- This function allows saving properties only for the current package or installable unit. It does not take the package name as a parameter and does not allow modifying properties of other packages.

SetPackageProperty

```
prototype SetPackageProperty(STRING, STRING);
```

Sets the property of package or installable unit.

Arguments

Field	Type	Description
1	STRING (input)	Property name.
2	STRING (input)	Property value.

Return Values

None

VersionToMajorMinor

```
prototype VersionToMajorMinor(STRING, BYREF NUMBER, BYREF NUMBER);
```

Converts version string to numeric values.

Arguments

Field	Type	Description
1	STRING (input)	Version number in string format (major.minor).
2	NUMBER (output)	Major version as a number.
3	NUMBER (output)	Minor version as a number.

Return Values

Value	Description
0	If the string does not contain a valid version, the returned major and minor versions are both set to 0.
-1	Component is not found.

VersionToMajorMinorPatch

```
prototype VersionToMajorMinorPatch (STRING, BYREF NUMBER, BYREF NUMBER, BYREF NUMBER);
```

Converts patch version string to numeric values.

Arguments

Field	Type	Description
1	STRING (input)	Version number in string format (major.minor.patch).
2	NUMBER (output)	Major version as a number.
3	NUMBER (output)	Minor version as a number
4	NUMBER (output)	Patch version as a number

Return Values

Value	Description
0	If the string does not contain a valid version, the returned major and minor versions are both set to 0.
-1	Component is not found.

IsVersionInRange

```
prototype IsVersionInRange(STRING, STRING, NUMBER);
```

This function checks if the specified package version is within the specified range.

Arguments

Field	Type	Description
1	STRING (input)	Component tag

Field	Type	Description
2	STRING (input)	The range, specified as M.m.p-M.m.p. Any lower part of a version can be omitted.
3	NUMBER (input)	Installed/pending flag: • RM_PKGPROP_PENDING—Package installed or package on CD. • RM_PKGPROP_INSTALLED—Previous package installed. • RM_PKGPROP_ANY—Package with latest installation version. • RM_PKGPROP_CD—Package on CD.

Return Values

Value	Description
0	The version is within limits.
-1	The version is lower than the lower limit.
1	The version is higher than the higher limit.
-99	The specified version was not found.

isPackagePending

```
prototype isPackagePending (STRING);
```

This function checks if the specified package is selected for installation.

Arguments

Field	Type	Description
1	STRING (input)	package tag

Return Values

Value	Description
TRUE	The specified package is selected for installation.
FALSE (0)	The specified package is *not* selected for installation.

Controlling Responses to Terminated Installations

Use the SetAbortFlag API to control a response to a terminated installation.

SetAbortFlag

```
prototype SetAbortFlag(STRING);
```

SetAbortFlag allows your hook to terminate installation with an error message. Use in preinstalls only. Do not abort after user completes installation because that would leave the product in an unknown state. If you believe that some serious problem has occurred during postinstall, send a message (MessageBoxLog function) and proceed.

Arguments

Field	Type	Description
1	STRING (input)	Log file entry

Processing Name=Value Pairs

Use this set of APIs to process files containing name=value pairs , similar to Java property files:

- addProperty
- addNumProperty
- loadPropertyFile
- addStringPropertyToList

addProperty

```
prototype addProperty(STRING, STRING, STRING, STRING);
prototype getProperty(STRING, STRING, STRING, BYREF STRING);
```

Adds or retrieves property to and from file.

Arguments

Field	Type	Description
1	STRING (input)	Directory
2	STRING (input)	File name
3	STRING (input)	Property name
4	STRING (input or output)	Property value: • Input for addProperty • Output for getProperty

Return Values

- addProperty—Searches for a specified property in the file. If found, it replaces the line; otherwise a new line is appended. If this file does not exist, it is created.
- getProperty—Returns 0 if property is found successfully.

addNumProperty

```
prototype addNumProperty(STRING, STRING, STRING, NUMBER);
prototype getNumProperty(STRING, STRING, STRING, BYREF NUMBER);
```

Adds or retrieves property to and from file.

Arguments

Field	Type	Description
1	STRING (input)	Directory
2	STRING (input)	File name
3	STRING (input)	Property name
4	NUMBER (input or output)	Property value: • Input for addNumProperty • Output for getNumProperty

Return Values

- addNumProperty—Searches for a specified property in the file. If found, it replaces the line; otherwise a new line is appended. If this file does not exist, it is created.
- GetNumProperty—Returns 0 if property is found successfully.

loadPropertyFile

```
prototype loadPropertyFile(STRING, STRING, LIST);
prototype savePropertyFile(STRING, STRING, LIST);
```

These functions allow you to preload properties from a file into a string list to speed up processing.

Arguments

Field	Type	Description
1	STRING (input)	Directory. Do not use the string list directly. Instead, use addProperty, addNumProperty, or addStringPropertyToList to set or retrieve properties.
2	STRING (input)	File name
3	LIST (input or output)	String list: • Input list for saveProperty • Output list for loadProperty

addStringPropertyToList

```
prototype addStringPropertyToList (LIST, STRING, STRING);
prototype addNumPropertyToList (LIST, STRING, NUMBER);
prototype getStringPropertyFromList (LIST, STRING, BYREF STRING);
prototype getNumPropertyFromList (LIST, STRING, BYREF NUMBER);
```

These functions are similar to add…/get… functions described previously.

Arguments

Field	Type	Description
1	LIST (input)	String list
2	STRING (input)	Property name
3	STRING or NUMBER (input or output)	Property value: • Input for addStringPropertyToList and addNumPropertyToList. • Output for getStringPropertyFromList and getNumPropertyFromList.

Sending Informational Messages to a Log File

The following APIs enable you to send messages to a log file:

- WriteLogFile
- AskYesNoLog
- AskYesNoLogTitle
- MessageBoxLog
- MessageBoxLogTitle
- StringListLog
- AddFileToLog

The log file is for information purposes only; it is not used for uninstallation.

WriteLogFile

```
prototype WriteLogFile(STRING);
```

Writes the string into the log file. Names and locations for these log files use the convention *%SystemDrive%*\CW2000_in*XXX*.log, where *XXX* is equal to the log sequence (001, 002, and so on).

Arguments

Field	Type	Description
1	STRING (input)	Message to write to log file.

AskYesNoLog

```
prototype AskYesNoLog(STRING,NUMBER);
```

Displays the AskYesNo dialog and puts both the dialog message and user's reply into the log file.

Arguments

Field	Type	Description
1	STRING (input)	Message. The same as InstallShield's AskYesNo function.
2	NUMBER (input)	Default answer.

AskYesNoLogTitle

```
prototype AskYesNoLogTitle (STRING,STRING,NUMBER);
```

Specifies the AskYesNo dialog title. This is the same as InstallShield's AskYesNo function (info, warning, error).)

Arguments

Field	Type	Description
1	STRING (input)	Dialog box title.
2	STRING (input)	Message. The same as InstallShield's AskYesNo function.
3	NUMBER (input)	Default answer

MessageBoxLog

```
prototype MessageBoxLog(STRING,NUMBER);
```

Displays the MessageBox dialog and writes a message into the log file. (The same as the InstallShield's MessageBox function.)

Arguments

Field	Type	Description
1	STRING (input)	Message. The same as InstallShield's AskYesNo function.
2	NUMBER (input)	Severity level. Predefined values: INFORMATION, WARNING, SEVERE

MessageBoxLogTitle

```
prototype MessageBoxLogTitle (STRING,STRING,NUMBER);
```

Specifies the MessageBox dialog title.

Arguments

Field	Type	Description
1	STRING (input)	Dialog box title.

Field	Type	Description
2	STRING (input)	Message
3	NUMBER (input)	Severity level. Predefined values: INFORMATION, WARNING, SEVERE.

StringListLog

```
prototype StringListLog(LIST, STRING);
```

Writes all elements of string list into log file.

Arguments

Field	Type	Description
1	LIST (input)	Prefix to be added to the beginning of each element.

AddFileToLog

```
prototype AddFileToLog(STRING, STRING);
```

Copies a file into log file.

Use this function to analyze results of processes executed using the Shell Task functions (see the "Running Commands in a Shell" section on page 21-52).

Arguments

Field	Type	Description
1	STRING (input)	Path name
2	STRING (input)	Title

Informing the Installer That a Component Requires More Space

The following APIs extract the disk cluster size and inform the installer that your component requires more space than its runtime footprint:

- GetClusterSizeEx
- needMoreSpace

GetClusterSizeEx

```
prototype GetClusterSizeEx(STRING, BYREF NUMBER);
```

Determines the size of cluster for specified path name.

Arguments

Field	Type	Description
1	STRING (input)	Valid path name
2	NUMBER (output)	Number to return cluster size

needMoreSpace

```
prototype needMoreSpace (STRING, NUMBER, NUMBER);
```

Informs installer that the component requires more space than its runtime footprint. Used only in preinstall hooks.

Arguments

Field	Type	Description
1	STRING (input)	Path name
2	NUMBER (input)	Required size in bytes
3	NUMBER (input)	Number of files

Registering and Unregistering CWCS Daemons

The following APIs enable you to register and unregister processes with the CWCS Daemon Manager:

- DmgrRegister
- DmgrUnregister

DmgrRegister

```
prototype DmgrRegister (STRING,INT,STRING,STRING,STRING);
prototype DmgrRegisterEx (STRING,INT,STRING,STRING,STRING,STRING);
prototype DmgrRegisterWR (STRING,INT,STRING,STRING,STRING,STRING,INT);
```

Registers a process with the CWCS Daemon Manager.

Arguments

Field	Type	Description
1	STRING (input)	Name
2	NUMBER (input)	Flag to run automatically or not. TRUE if daemon should be started automatically.
3	STRING (input)	Executable path name. All processes need to be registered with the default path (/opt/CSCOpx on Solaris) explicitly instead of $NMSROOT.
4	STRING (input)	Arguments (separated by a caret).
5	STRING (input)	Daemon Manager's dependencies. A comma-separated list of other daemons on which this daemon depends.

Field	Type	Description
6	STRING (input)	A flag to run in local system account. • 1 = Daemon should run in LocalSystem account. • 0 = Run as casuser.
7	NUMBER (input)	Maximum amount of time (in milliseconds) that Daemon Manager should wait for the process to initialize.

For cwjava daemons, use the Java versions of these APIs: `DmgrRegisterJava`, `DmgrRegisterJavaEx`, and `DmgrRegisterJavaWR`. These Java calls share the C syntax and arguments.

DmgrUnregister

```
DmgrUnregister (STRING);
```

Unregisters a daemon from the CWCS Daemon Manager. Use this function with the uninstall hook to undo registration.

Arguments

Field	Type	Description
1	String	Name of the daemon to unregister. This should match the Name parameter of the DmgrRegister, DmgrRegisterEx, or DmgrRegisterWR command used to register it.

Running Commands in a Shell

The following APIs enable you to run specified commands in a shell. The shell window is minimized. Control returns after command line task finishes.

- LaunchShellAndWait
- InitBatchFile

LaunchShellAndWait

```
prototype LaunchShellAndWait(STRING, STRING, BYREF STRING, BYREF STRING);
```

Runs the specified command and minimizes the shell window. It waits until the specified task is completed.

Arguments

Field	Type	Description
1	STRING (input)	Job Title. This title must be unique. It will be displayed in the task bar.
2	STRING (input)	Command line.
3	STRING (output)	File to redirect standard output. Specify as variables, not as constants.
4	STRING (output)	File to redirect standard error. Specify as variables, not as constants.

Notes

- Output files can be specified as empty, in which case standard output and standard error are not redirected. To run a batch file, use InitBatchFile (see the "InitBatchFile" section on page 21-53) and RunBatchAndWait (see the "RunBatchAndWait" section on page 21-53) .
- Either both or one of the output files can be an empty string. In this case, standard output or standard error will not be captured; it will be ignored.
- If the file name is specified, it must be a filename without a path. After this function completes, the variable of the corresponding file will contain the full path name for an output directory.

InitBatchFile

```
prototype InitBatchFile(STRING, STRING, STRING, BYREF NUMBER);
```

Initializes a batch file to execute by RunBatchAndWait command.

Arguments

Field	Type	Description
1	STRING (input)	Job title. This title must be unique. It will be displayed in the task bar.
2	STRING (input)	Command line.
3	STRING (input)	Temporary file name.
4	NUMBER (output)	Number to return open file handle. Used to insert more lines into batch file.

Notes

For RunBatchAndWait (see the "RunBatchAndWait" section on page 21-53) to work properly, the batch file must be initialized by InitBatchFile. Several lines can be added by InstallShield's WriteLine function and closed by the CloseFile function.Then run the RunBatchAndWait command using the same first and second parameters as specified for InitBatchFile.

RunBatchAndWait

```
prototype RunBatchAndWait(STRING, STRING, BYREF STRING, BYREF STRING);
```

Runs the batch file created by InitBatchFile.

Arguments

Field	Type	Description
1	STRING (input)	Job title. Must be the same title as specified for the InitBatchFile function.
2	STRING (input)	Temporary file name. Must be the same name as specified for the InitBatchFile function.
3	STRING (output)	Name of file to redirect standard output. Specify as variables, not as constants.
4	STRING (output)	Name of file to redirect standard error. Specify as variables, not as constants.

Notes

- For this function to work properly, the batch file must be initialized by InitBatchFile (see the "InitBatchFile" section on page 21-53). Several lines can be added by InstallShield's WriteLine function and closed by CloseFile function. Then run the RunBatchAndWait command using the same first and second parameters as specified for InitBatchFile.
- Either both or only the last one can be an empty string. In this case, standard output or standard error will not be captured. It will be ignored. If the filename is specified, it needs to be just a filename without a path. After this function is finished, the variable of the corresponding file will contain the full path name for an output directory.

Locating the Root Directory Path Name

The GetRootDir function allows you to find the path name for the root directory.

GetRootDir

```
prototype GetRootDir(STRING, BYREF STRING);
```

Find the path name for the root directory.

The installer sets actual directory path names at runtime:

1. The first time, the installer collects values of this property from all packages and installable units and asks the user to assign each one an actual path name.
2. The installer saves these values in the Windows registry.

   The next time the installer runs, it gets the path names from the registry, requiring no input from the user. This allows multiple root directories, for example, one for CWCS and Essentials (*NMSROOT*) and another one for Connectivity suite (for example, *CONNROOT*).
3. The installer asks the user for path names for both directories and makes them available programmatically.

   In package hooks, the value of ROOTDIR property for this package can be used directly as a global variable. If a hook needs a value of rootdir for another component, it can be obtained using the GetRootDir function.

Arguments

Field	Type	Description
1	STRING (input)	Root directory name.
2	STRING (output)	Path name of root directory.

Return Values

Value	Description
0	If specified root directory found.

Notes

The root directory path name for the current component is set to a global variable and can be used directly. This function is needed to obtain the path name of the root directory of another component. For example, package MYPKG has the property ROOTDIR=*MYROOTDIR*.

Example 21-1 Using GetRootDir

```
Function mypkg_postinstall()
STRING mdPropertyPath, nmsRoot;
STRING myPkgPropFileName;
begin
    // property file for this package is under its root directory and
    // MYROOTDIR can be used directly
   myPkgPropFileName = MYROOTDIR^"myPropFile.properties";
   …
   // md.property file is somewhere under daemon manager's root - NMSROOT
   if (GetRootDir("NMSROOT", nmsRoot) = 0) then
       mdPropertyPath = nmsRoot^"subdir1\\…";
       …
   endif
…
end;
```

Registering and Controlling Windows Services

CWCS provides a set of APIs and constants for registering and controlling Windows services. Use the following service APIs to register and control Windows services:

- RegisterService
- UnregisterService
- StartService
- ChangeServiceStartType
- ChangeServiceAccount
- ChangeService2Casuser
- StopService

Using Windows Service Constants

Whenever you use the service APIs, use the following constants to specify service types:

- SERVICE_WIN32_OWN_PROCESS
- SERVICE_WIN32_SHARE_PROCESS
- SERVICE_INTERACTIVE_PROCESS

You can also use the following constants to specify service start types:

- SERVICE_BOOT_START
- SERVICE_SYSTEM_START

- SERVICE_AUTO_START
- SERVICE_DEMAND_START
- SERVICE_DISABLED

RegisterService

```
prototype RegisterService (STRING,STRING,STRING,LONG,LONG);
```

Registers new Windows Services.

Arguments

Field	Type	Description
1	STRING (input)	Service name
2	STRING (input)	Service display name
3	STRING (input)	Path name of executable
4	LONG (input)	Service type. Specify a service type constant (see the "Using Windows Service Constants" section on page 21-55).
5	LONG (input)	Service start type. Specify a service start type constant (see the "Using Windows Service Constants" section on page 21-55).

UnregisterService

```
prototype UnregisterService(STRING);
```

Unregisters Windows services.

Arguments

Field	Type	Description
1	STRING (input)	Service name

StartService

```
prototype StartService(STRING);
```

Starts Windows services immediately.

Arguments

Field	Type	Description
1	STRING (input)	Service name

ChangeServiceStartType

```
prototype ChangeServiceStartType(STRING, LONG);
```

Changes service start type.

Arguments

Field	Type	Description
1	STRING (input)	Service name
2	LONG (input)	Service start type. Specify a service start type constant (see the "Using Windows Service Constants" section on page 21-55).

ChangeServiceAccount

```
prototype ChangeServiceAccount (szServiceName, accountName,
    accountPassword);
```

Changes the service account. Use this function after the service is created (see the "RegisterService" section on page 21-56).

Arguments

Field	Description
szServiceName	The name of the service.
accountName	The account for the service to run as. To modify the service to run as LocalSystem account, the value should be ".\LocalSystem" (without quotes, and remember to escape the backslash).
accountPassword	The password for the account. Leave this empty if the account is LocalSystem.

Related Topics

See the "ChangeService2Casuser" section on page 21-57.

ChangeService2Casuser

```
prototype ChangeService2Casuser(STRING, POINTER);
```

This function reconfigures a service to run as casuser. This relieves the developer from the need to fetch the value of the casuser password, which is maintained by the Daemon Manager. This function can only be used after the Daemon Manager has been installed.

Arguments

Field	Type	Description
1	String	The name of the service.
2	Pointer	The pointer to the string that contains the password. Specify NULL to use the password stored by the Daemon Manager.

Return Values

0 if successful.

Example

This function is implemented in secure.dll, which is a part of the installation framework. Therefore, a call to this function must be framed with the UseDll/UnUseDll calls. For example:

```
if (UseDLL(SUPPORTDIR ^ "secure.dll") != 0) then
    MessageBoxLog("Cannot load secure.dll", SEVERE);
endif;
ChangeService2Casuser("myService", NULL);
UnUseDll(SUPPORTDIR ^ "secure.dll");
```

StopService

```
prototype StopService(STRING, NUMBER);
```

Stops service.

Arguments

Field	Type	Description
1	STRING (input)	Service name
2	NUMBER (input)	Stop mode. Options: • RM_STOPSERV_IMMEDIATELY • RM_STOPSERV_DELAYED

Notes

- In preinstalls, use RM_STOPSERV_DELAYED. The installer does not stop services immediately; instead, it collects the list of services to stop.
- After all preinstalls are executed, the installer asks the user for confirmation and stops all requested services at once. In addition, the installer stops all services that were requested to be stopped.
- In all hooks other than preinstall, use the RM_STOPSERV_IMMEDIATELY mode.

Using Generic Utilities

The generic installation framework utilities contain functions for string list processing, path processing, deleting files, displaying system error messages, and obtaining DNS names:

- EmptyList
- InvertList
- GetStringFromList
- ListFindSubstring
- AddDirToPath
- DeleteFromPath
- pathToFS
- ShowLastSysError
- SureDeleteFile
- GetHostName

- ModifySubParameter

EmptyList

```
prototype EmptyList(LIST);
```

Removes all elements from the list.

Arguments

Field	Type	Description
1	LIST (input and output)	Name of valid string list. Input list will be modified.

InvertList

```
prototype InvertList(LIST);
```

Changes the order of elements in a string list.

Arguments

Field	Type	Description
1	LIST (input and output)	Name of valid string list. Input list will be modified.

GetStringFromList

```
prototype GetStringFromList(LIST, NUMBER, BYREF STRING);
```

Retrieves an element from a string list by number. For example, it will return the third string in a list.

Arguments

Field	Type	Description
1	LIST (input)	Name of valid string list
2	NUMBER (input)	Index
3	STRING (output)	String returned by the function. It is “ ” if the requested string is not available.

ListFindSubstring

```
prototype ListFindSubstring(LIST, STRING, NUMBER, BYREF STRING);
```

Finds an element in a string list, which contains given string starting from the current element.

Arguments

Field	Type	Description
1	LIST (input)	Name of valid string list.
2	STRING (input)	String to search for.
3	NUMBER (input)	Offset. –1 stands for any offset.
4	STRING (output)	String returned by the function. The same as ListGetNextString. It is “ ” if the requested string is not available.

Notes

- InstallShield’s ListFindString searches for an element that is exactly similar—it does not allow searching for a substring.
- The third parameter allows you to specify an offset. For example, to find an element that begins with “ABC”, call ListFindSubstring(listID, “ABC”, 0, svValue).

AddDirToPath

```
prototype AddDirToPath(STRING);
```

Adds a specified directory to the path.

Arguments

Field	Type	Description
1	STRING (input)	Path name of directory to be added

DeleteFromPath

```
prototype DeleteFromPath(STRING);
```

Deletes path name from the path.

Arguments

Field	Type	Description
1	STRING (input)	Path name to be removed.

pathToFS

```
prototype pathToFS(BYREF STRING);
prototype pathToBackSlash(BYREF STRING);
```

Converts backslashes in path name into forward slashes and vice versa.

Arguments

Field	Type	Description
1	STRING (input)	String that contains path name. After execution, the modified path name.

ShowLastSysError

```
prototype ShowLastSysError(STRING, LONG);
```

Display system error message in dialog box.

Arguments

Field	Type	Description
1	STRING (input)	Title of dialog box
2	LONG (input)	Error code

Example

```
ShowLastSysError ("OpenSCManager failure",GetLastError());
```

SureDeleteFile

```
prototype SureDeleteFile(STRING);
```

Removes specified file.

Arguments

Field	Type	Description
1	STRING (input)	Installer of a file to remove. Processes both short and long file names

GetHostName

```
prototype GetHostName(BYREF STRING);
```

Retrieves the DNS host name.

Arguments

Field	Type	Description
1	STRING (output)	String variable to receive host name.

ModifySubParameter

```
prototype ModifySubParameter(BYREF STRING, STRING, STRING, BOOL, STRING, STRING);
```

Modifies the value of a subparameter in a command line or removes it.

Arguments

Field	Type	Description
1	STRING (output)	String to modify.
2	STRING (input)	Parameter whose value you want to modify.
3	STRING (input)	New subparameter or subparameter to remove.
4	BOOL (input)	TRUE to add or replace subparameter; FALSE to remove subparameter.
5	STRING (input)	Delimiters for parameters
6	STRING (input)	Delimiters for subparameters

Notes

For example, you can modify the line:

```
myExe.exe -d p1,p2,p3 -p jhgjhg -t
```

to look like

```
myExe.exe -d p1,p2,p3,newP -p jhgjhg -t
```

by calling

```
ModifySubParameter(myStr, "-d", "newP", TRUE, " ", " ,")
```

where `myStr` has a string to modify.

Managing Passwords

In this release, there are two sets of functions that manage passwords:

- The functions AskPass and GetPass provide CMF 2.1-style password management, where the user can modify the value of the default password.
- In CWCS 2.2, the functions SetPassEx, GetPassEx, and AskPassEx facilitate adding password-related dialogs as a part of the installation user interface.

This section contains the password management APIs:

- AskPass
- GetPass
- AskPassEx, AskPassExTitle
- GetPassEx
- SetPassEx
- ChangeDbPasswd.pl

AskPass

```
AskPass (Prompt_String, Temp_File_Name, DbPass, Password)
```

The AskPass API:

1. Prompts for confirmation to change the password. Enter Yes or No.
   - If you enter Yes, AskPass prompts for the new password.

   – Enter the new password, confirm it, and click **Next**.

2. Validates the password and stores it in a temporary file.
3. Encrypts the file using the cipher command.

Arguments

Field	Type	Description
Prompt_String	String	The confirmation prompt.
Temp_File_Name	String	Name of the temporary file where the password is stored.
DbPass	Integer	Used to validate the password. Whenthe value of this argument is *1*, the password is validated according to database rules. For example: • The password should not start with a number. • The password should not contain any special characters and Password length should not be more than 15 characters. When set to values other than 1, no validation is done.
Password	String	Displays the user-entered or randomly-generated password.

Notes

This API is called in the preinstall function of the Db.rul.

Example

```
AskPass("Enter Common Services Database Password",
    "Changing Database Password",
    "enpass_cmf", 1);
```

GetPass

```
GetPass(Temp_File_Name, Password)
```

Use this function to retrieve the value saved by the AskPass function. The GetPass API:

1. Retrieves the password from the temporary file.

   GetPass accepts the file name and a string variable as the parameter.
2. Decrypts the temporary file and returns the password in the variable.
3. Deletes the file once the password is read.

   GetPass returns 0 if it succeeds in getting the password, and 1 if it fails.

Arguments

Field	Type	Description
Temp_File_Name	String	Name of the file where the password is stored.
Password	String	Variable where the password will be returned

Example

```
if (GetPass("casuser_pass",passwd) = 0) then
...
```

In this example, the API reads the password from the file casuser_pass. The variable passwd contains the password.

AskPassEx, AskPassExTitle

```
prototype AskPassEx (szDescription, szComment, dialogId, szDLL,
    fileName, mode, lLabels, nIdStartPwd);
prototype AskPassExTitle (szDescription, szComment, dialogId, szDLL,
    fileName, mode, lLabels, nIdStartPwd, szTitle);
```

These functions prompt the user to specify a new password or accept the default or existing password. The value of the password is stored at a temporary location in encrypted format and can be retrieved by the GetPassEx API (see the "GetPassEx" section on page 21-65). Use this function in the custom panel. For details and a sample, see the "Customizing the Installation Workflow for Windows" section on page 21-76.

Arguments

Argument	Description
szDescription	The text of description, displayed above the input fields.
szComment	The text of the comment, displayed below the input fields.
dialogId	ID of the dialog template. Currently, the installation framework contains two templates: • 13039 (with input fields for one password with confirmation) • 13034 (with input fields for two passwords with confirmations).
szDLL	The name of the DLL that contains the dialog templates. Use an empty string for templates supported by the installation framework.
fileName	The name of temporary file that stores the values for the duration of installation. Do not use path names.
mode	Password verification mask. Can be a combination of the following: • PASS_EX_ANY—Any value is allowed (should not be combined with any other value). • PASS_EX_MAX_LENGTH—The maximum length of the password. • PASS_EX_NONEMPTY—The password must be not empty (should not be combined with PASS_EX_EMPTY_ALLOWED). • PASS_EX_EMPTY_ALLOWED—The empty password is allowed (should not be combined with PASS_EX_NONEMPTY). • PASS_EX_ALPHANUM—The password can only contain digits and alphabetic characters. • PASS_EX_FIRST_ALPHA—The first character must be alphabetic. • PASS_EX_MIN_LENGTH—The minimum length of the password. The example illustrates some of these options.
lLabels	List of the labels of input fields. Should contain labels for fields, followed by the labels of confirmation fields (optional). By default, the confirmation fields are labelled "Confirm"/.

Argument	Description
nIdStartPwd	ID of the first input field. For the dialogs supported by the framework, use 0.
szTitle	(In AskPassExTitle only)—The title of the dialog. The dialog displayed by AskPassEx uses the title "Change Password""

Return Codes

NEXT or BACK, depending on whether the user clicked the Next or Back buttons.

Example

The following code fragment displays a dialog that lets the user specify values for two passwords (with confirmation for each of them), and stores the result in the admin_pass file in a temporary location. Both passwords cannot be empty, must be at least 5 characters long, contain only alphanumeric characters, and the first character must be alphabetical.

```
lLabels = ListCreate(STRINGLIST);
ListAddString(lLabels, "User admin Password", AFTER);
ListAddString(lLabels, "User guest Password", AFTER);
nRc = AskPassExTitle("You may change the password of admin and guest users.\n" +
        "Leave fields empty to keep existing passwords.",
    "Password must begin with an alphabetic character and should be "+
        "at least 5 characters long.",
    13034,"13034,"", "admin_pass",
    PASS_EX_NONEMPTY + PASS_EX_ALPHANUM +
        PASS_EX_FIRST_ALPHA + PASS_EX_MIN_LENGTH * 5,
    lLabels, 0, "Change Admin and Guest Password");
ListDestroy(lLabels);
```

Related Topics

See the:

- "GetPassEx" section on page 21-65.
- "Customizing the Installation Workflow for Windows" section on page 21-76.

GetPassEx

```
prototype GetPassEx (szFileName, svField, nField);
```

This API retrieves the values stored by AskPassEx or SetPassEx.

Arguments

Name	Description
fileName	File name. Must match the fileName parameter of the AskPassEx or SetPassEx functions.
svField	The variable that receives the value.
nField	The field number (counting from 0).

Return Values

Value	Meaning
0	Successful
1	Not successful

SetPassEx

```
prototype SetPassEx (lData, fileName);
```

Stores values in the temporary location in an encrypted format. This is suitable to pass values from early stages of installation to later stages. Encryption will only work on a Windows operating system that supports encryption.

Arguments

Name	Description
lData	List of values to be preserved.
fileName	The name of temporary file that stores the values for the duration of the installation. Do not use path names.

Return Values

Value	Meaning
0	Successful
1	Unsuccessful

Notes

Subsequent calls with the same fileName parameter will replace the previously stored values.

ChangeDbPasswd.pl

```
$NMSROOT\bin\perl.exe
$NMSROOT\objects\db\conf\ChangeDbPasswd.pl DSN New_Password
```

This API changes the password in the database and property files. It accepts the DSN name and password as the parameters and changes the password.

Note This API is normally called in the postinstall function of the DB package.

The ChangeDbPasswd.pl API:

- Checks whether the DSN is valid.
- (If the DSN is valid) checks whether the database is enabled,
- (If the database is enabled), changes the password in the database, odbc.tmpl, .odbc.ini, DBServer.properties and the property file specified in .odbc.tmpl.
- (If the database is not enabled), changes the password in the database and odbc.tmpl.

Arguments

Field	Type	Description
DSN	string	Database Name
Password	string	New Password

Return Values

Value	Description
0	Password was changed successfully.
>0	Password change failed.

Example

```
$NMSROOT\bin\perl.exe
$NMSROOT\objects\db\conf\ChangeDbPasswd.pl cmf cisco
```

Configuring Tomcat

The Tomcat configuration APIs include:

- ModifyFolderXML
- UpdateTomcat

ModifyFolderXML

```
ModifyFolderXML(fileName, replacedString, contextPath, finalString)
```

Configures the application Desktop XML file to integrate it with CWCS.

Arguments

Field	Description
fileName	Required. The folder file you want to integrate.
replacedString	Required. The placeholder to be replaced with CWCS URL information.
contextPath	Required. The URL pattern that identifies your servlet context.
finalString	Optional. The string to use as a replacement within your file. The default string provided by CWCS is based on data from CCR regarding the server name, protocol, and port number.

Notes

This method delegates its work to a MICE installation utility class. The actual command line that will be run looks like this:

```
<path to java>/java com.cisco.core.mice.util.install.FolderXMLModifier <Absolute Path to
XML> <String to replace> <context url pattern> <String to prepend>
```

Example

In this example,

```
java com.cisco.core.mice.util.install.FolderXMLModifier
    C:/MyApp.xml REPLACE_THIS_STRING /myApp
```

This command will replace all instances of the string REPLACE_THIS_STRING in the drawer file with the default prepend value from CCR, which consists of the URL values to talk to Common Services and initialize the session data between CiscoWorks applications and Common Services.

The ModifyFolderXML utility can transform this line fragment in the myApp.xml file from this:

```
<item NAME="Launch myApp"
    HREF="REPLACE_THIS_STRING/myapp?command=firstPage">
```

to this (all one line):

```
<item NAME="Launch myApp"
    HREF="/CSCOnm/servlet/com.cisco.core.mice.util.cmf.
    CMFLIaisonServlet?command=initializeAndValidate&amp;
    url=%2Fmyapp%3Fcommand%3DfirstPage&amp;
    context=/myapp&amp;port=443">
```

This method:

- Sets the URL up to bounce to the CMFLiaisonServlet on the Tomcat servlet engine.
- Passes information to that servlet about what port the Common Services Tomcat engine is running on, and what servlet context the eventual URL target is going for.
- URLEncodes the original URL to preserve whatever data or parameters an application is attaching to their requested URL.

UpdateTomcat

```
UpdateTomcat (aUrlPattern, aRelativePath, aMapping, appid)
UninstallUpdateTomcat (aUrlPattern, aRelativePath, aMapping, appid)
```

Call the `UpdateTomcat` method when you are finished populating CCR (the CWCS Client Registrar). This method:

Handles the configuration of Apache and Tomcat.

- Populates CCR with the information required by the application servlet context to operate within the core multi-context environment.
- Adds the JkMount directive to httpd.conf.
- Adds the <Context> node into the apps-ordered.xml file of Tomcat.
- Creates a custom ContextInfo entry in CCR.

The UninstallUpdateTomcat method cleans up data related to the application.

Both methods perform some configuration, but delegate most of the work to the Java utility class TomcatServiceUpdate. This class is called from the command line within the installation framework.

Arguments

Field	Description
aUrlPattern	The application's context path.
aRelativePath	The application's docbase relative to Tomcat.
aMapping	The servlet mapping of the application's CWCS liaison servlet.
appid	The ID that identifies this application to CAM.

Notes

All arguments must be defined for full integration and interaction to occur with CWCS.

Example

This is a custom ContextInfo entry in CCR:

```
<Custom22>
          <Name>ContextInfo</Name>
          <Location>/testLiaison</Location>
          <Data>/myApp</Data>
          <appid>myApplication</appid>
          <ReferenceCount>1</ReferenceCount>
          <References>
                    <Core />
          </References>
</Custom22>
```

Controlling Reboots

The SetRebootFlag function allows you to request a reboot at the end of an install or uninstall.

SetRebootFlag

```
SetRebootFlag()
```

Causes the framework to request a system reboot from the user at the end of an installation or uninstallation.

Arguments

None.

Notes

This method tells the installation framework that the system should be rebooted once the installation or uninstallation is finished. The framework will open a dialog requesting the user to confirm the reboot. The user can choose to either reboot at the time of the prompt or to “reboot later”.

You can use SetRebootFlag in a Preinstall or Postinstall to request reboot at the end of installation, or in an Uninstall to reboot after uninstallation.

You can also request reboot at the end of uninstallation by adding `UNINSTALL_REBOOT=Y` to the package properties file (see Table 21-5 on page 21-7).

Example 21-2 Using GetRootDir

```
Function mypkg_postinstall()
STRING mdPropertyPath, nmsRoot;
STRING myPkgPropFileName;
begin
    // property file for this package is under its root directory and
    // MYROOTDIR can be used directly
   myPkgPropFileName = MYROOTDIR^"myPropFile.properties";
   …
   // md.property file is somewhere under daemon manager's root - NMSROOT
   if (GetRootDir("NMSROOT", nmsRoot) = 0) then
       mdPropertyPath = nmsRoot^"subdir1\\…";
       …
   endif
…
end;
```

Using Windows Build Tools

This topic provides instructions for building an image from protopackages using the installation framework. The main steps are:

- Step 1: Install Third-Party Tools for Windows
- Step 2: Install the Framework on Windows Platforms
- Step 3: Prepare the Make Image on Windows Platforms

The following topics provide additional guidelines and examples:

- Debugging on Windows Platforms
- Example: Using Windows Build Tools

For examples showing how to add your application to the CWCS image, see the "Solaris Getting Started Example" section on page 21-106.

Step 1: Install Third-Party Tools for Windows

Install the Windows third-party tools listed in the "Third-Party Tools for Installation Framework" section on page 21-4.

Step 2: Install the Framework on Windows Platforms

To install the framework, copy the following files to your Windows hard drive:

- buildImage: This main script creates the CD image from protopackages.
- verifyImage: This script verifies the structure of the protopackages.
- is5.runtime.tar: This protopackage contains the Windows installation framework.

Step 3: Prepare the Make Image on Windows Platforms

Follow these steps to prepare the make image on Windows platforms:

Step 1 Verify that MKS is in your path, by entering the following commands:

```
which sh
which perl
which find
```

The returned path names for shell, perl, and find should refer to the MKS tools. If this is not true, modify the path to include MKS binaries.

Step 2 If InstallShield or PackageForTheWeb are installed in non-default directories, set the following environment variables:

MK_IS55=*full path where InstallShield is installed*

MK_PFTW=*full path where PackgeForTheWeb is installed*

For both environment variables use backslashes with short path name. Spaces are not allowed. For example

```
set MK_IS55=d:\progra~1\instal~1\instal~1.5pr CORRECT
set MK_PFTW=d:\progra~1\instal~1\packag~1 CORRECT
set MK_IS55= D:\Program Files\InstallShield\InstallShield 5.5 Professional Edtn INCORRECT
```

Note The short path name can be specific to your system. To determine the short path name, use the command **dir /x.**

Step 3 Run the buildImage command:

```
perl buildimage -r -d image_path rtpath/is5.runtime.tar pkgpath/myPkg.runtime.tar
protopackage…
```

Where:

- *-r* is an option to refresh the *image_path* area by first removing all directories from *image_path*\extract and then extracting all protopackages.
- *image_path* is the full path name of a directory where the image will be created. A lot of free space is required on this drive, at least three times the size of the runtime. You must specify the path starting from the drive letter, with forward slashes, no spaces allowed.
- *rtpath* and *pkgpath* are the full path names of the directories where the is5.runtime.tar and you package are stored, respectively
- *protopackage* should be specified with full path names with forward slashes, no spaces allowed. For the complete list of protopackages, see the sample scripts below.

This command will create the following directories under *image_path*:

- extract: This contains all files extracted from protopackages.
- temp: This contains temporary files required to arrange an image.
- disk1: This contains the installable image.
- *ReleaseName*.exe: This is the self-expanding installable image. *ReleaseName* will be similar to the name of the release as specified in the table of contents.

Debugging on Windows Platforms

After the buildImage script has created the extract directory, you do not have to extract protopackages again. Instead, you can change into the extract\is5\install directory and run the *runme.sh* script:

```
cd myImage_path\extract\is5\install
sh runme.sh > myImage_path\my.log 2>&1
```

You can apply changes directly to the files under the extract directory. Do not forget to put changes back into corresponding protopackages.

InstallShield's debugging information is available under image_path\temp\isproject\Script Files*.dbg. To run installation hooks in the debugger, copy the corresponding .dbg file to the disk1 directory with the image and run installation from the command line with the DEBUGHOOKS parameter. For example:

```
cd myImage_path\disk1
copy ..\temp\isproject\Script files\tag.dbg
setup DEBUGHOOKS
```

Where *tag* is the tag of the protopackage you want to debug.

Installation will run preinstall and postinstall scripts in the debugger. It can also open message boxes complaining about .dbg files not being available for other protopackages. Click OK and proceed with the installation.

The installation framework contains a batch file, rul.cmd, that simplifies the development process on Windows. This file:

- Is located in the is5.runtime.tar in the is5\install directory. The buildImage command creates the extract directory, as well as the installable image.
- Lets you modify the hooks directly in the extract area, and then immediately recompile them into the image.
- Requires the IMAGE environment variable. It should be set to the pathname of the directory that contains the extract area. For example, if the extract area was created in the d:\imagePath directory, specify the IMAGE environment variable like this:

  ```
  set IMAGE=d:\imagePath
  ```

To recompile the code for the hook of *myApp* package, modify the code in the d:\imagePath\extract\myApp\install\myApp.rul file, then run the following command:

```
d:\imagePath\extract\is5\install\rul.cmd myApp
```

This command recompiles the code directly into d:\imagePath\disk1 and copies the debugging information (myApp.dbg file).

Example: Using Windows Build Tools

The following topic provides examples on how to create an installable CWCS CD image with a customer application.In these examples, we assume the following:

- The *myapp* name refers to an application called *my application.* The application name is truncated to meet the Solaris requirement for five-letter package names (CSCOxxxxx).
- Our sample application is in two tar-files, myapp.cd.tar and myapp.runtime.tar, both in the current directory. These files are copied automatically by the installation script to the following target directories: myapp.cd.tar into myapp\disk1, and myapp.runtime.tar into myapp\install and myapp\runtime structures.

The directory myapp\disk1 contains the following disk.toc file describing our sample application's table of contents.

```
[RELEASE]
NAME=CWCS with Test Application
VERSTR=2.0
REGISTRY_ROOT=SOFTWARE\Cisco\Resource Manager\CurrentVersion

[COMPONENTS]
TAGS=cwcs cmfwd cmfj2 myapp
UNINSTALLABLE=cmfwd myapp
VISIBLE=cwcs cmfwd myapp
CHOICE=cwcs cmfwd myapp
DEFAULT=ALL

[ADVANCED_CHOICE_1]
ADVANCED_CHOICE_1_CONDITION=cmfwd.1.0.0-1.9.99
ADVANCED_CHOICE_1_TYPE=EXCLUSIVE
ADVANCED_CHOICE_1_DEFAULT=4
ADVANCED_CHOICE_1_1_TEXT=CiscoWorks Common Services (CWCS) Base Desktop
ADVANCED_CHOICE_1_1_TAGS=cmfwd
ADVANCED_CHOICE_1_2_TEXT=CWCS (including Base Desktop)
ADVANCED_CHOICE_1_2_TAGS=cwcs
ADVANCED_CHOICE_1_3_TEXT=myapp Application
ADVANCED_CHOICE_1_3_TAGS=myapp
ADVANCED_CHOICE_1_4_TEXT=myapp Application and CWCS
ADVANCED_CHOICE_1_4_TAGS=cwcs cmfwd myapp

[ADVANCED_CHOICE_2]
ADVANCED_CHOICE_2_CONDITION=TRUE
ADVANCED_CHOICE_2_TYPE=EXCLUSIVE
ADVANCED_CHOICE_2_DEFAULT=3
ADVANCED_CHOICE_2_1_TEXT=CiscoWorks Common Services (CWCS) Base Desktop
ADVANCED_CHOICE_2_1_TAGS=cmfwd
ADVANCED_CHOICE_2_2_TEXT=CWCS (including Base Desktop)
ADVANCED_CHOICE_2_2_TAGS=cwcs
ADVANCED_CHOICE_2_3_TEXT=myapp Application and CWCS
ADVANCED_CHOICE_2_3_TAGS=cwcs cmfwd myapp
```

For this toc file you need the following:

- REGISTRY_ROOT provides the key name for component information. The value has been set for the CWCS release and has to be the same to keep the compatibility with the CWCS.

Note The NAME parameter in the [RELEASE] section is used to make a name of an executable. All spaces are replaced by underscores, but parenthesis, comma slashes, and other characters are not allowed in filenames.

The ADVANCED_CHOICE section determines what is the correct scenario depending on what is already installed on the target machine. In the case where the target machine has a previous version of CWCS Desktop already installed, the customer has the following options:

- Upgrade to a new version of CWCS Desktop
- Install CWCS 3.0
- Install the application myapp on top of CWCS Base Desktop
- Install his application bundled with CMF 2.2.

In the case of a fresh installation, you don't have the option of a separate myapp installation. So in this case there are only the remaining three options mentioned in ADVANCED_CHOICE_2 section.

- The directory myapp\install contains the myapp.bprops and myapp.pkgpr files. The file myapp.bprops provides build identification. For example:

  ```
  D:\myapp\install>cat myapp.bprops
  PROP_ID = build_test
  PROP_TIMESTAMP = 939800833
  ```

  The myapp.pkgpr file contains package properties that specify the name, version, and dependencies. For example:

  ```
  D:\myapp\install>cat myapp.pkgpr

  NT:AIX:HPUX:SOL:
  NAME=Sample Package
  DESC=This is an example of installation framework
  VERSION=5.0
  PATCHVER=0
  DEPENDS=cmfwd

  SOL:
  PKG=CSCOmyapp

  NT:
  PKG=myapp
  ```

  The directory myapp\runtime contains the cgi-bin and htdocs subdirectories, corresponding to the structure of similar directories of the CiscoWorks-product to be used by the web-server. Two Perl scripts, cgi-bin\myappmyappcgi.pl and htdocs\myapp\myappframe.html file. Each developer must replace these files with their application files. Two files, htdocs\Xml\System\maintree\myapp.xml and htdocs\Xml\System\maintree\myappcgi.xml, are for linking our html-and Perl-files to the appropriate tree structure of the CiscoWorks Home Page. Each developer must swap their xml-files in their place. For the details on integrating your application with CiscoWorks, refer to the "Integrating Your Application with CWHP" section on page 7-6.
- The protopackages for CMF 2.0 are in the \...\protopackages directory of the main SDK kit.
- is5.runtime.tar file is in \...\protopackages directory of the main SDK kit.
- BuildImage file is in the root directory of the main SDK kit.
- InstallShield 5.53 is installed at C:\Program Files\InstallShield\InstallShield 5.5 Professional Edition and InstallShield PackageForTheWeb is installed at C:\ProgramFiles\InstallShield\PackageForTheWeb 2.
- The environment variables are set as:

  MK_IS55=c:\progra~1\instal~1\instal~1.5pr

  MK_PFTW=c:\progra~1\instal~1\packag~1
- The MKS tools are in the path (suppose C:drive is used):

  C:\>which sh

  C:\MKS\MKSNT/sh.exe

  C:\>which perl

  C:\MKS\MKSNT/perl.exe

  C:\>which find

  C:\MKS\MKSNT/find.exe
- InstallShield is installed properly.

Note You must make the corresponding changes in these variables, path, and InstallShield installation parameters.

To create the CWCS image, use the sample2.sh script in current directory.

- Launch the sample2.sh file using sh and with four arguments (don't use \ in args, use /):

  >sh ./sample2.sh *arg1 arg2 arg3 arg4*

  where:

  - *arg1* is the target directory; let it be for example D:/image directory
  - *arg2* is the directory with files myapp.cd.tar and myapp.runtime.tar (actually current directory)
  - *arg3* is the directory with buildImage file
  - *arg4* is the directory with protopackages and is5.runtime.tar files

  This is the sample2.sh file:

```
######################################################################
# INPUT:
# 1 - Target_dir
# 2 - myapp_dir
# 3 - buildImage_dir
# 4 - proto_dir protopackages&is5 directory
######################################################################

if [$# -ne 4]; then
    echo "ERROR:sample2 called with insufficient args."
else
    Target_dir=$1
    if [-d $Target_dir]; then
        echo "Directory $Target_dir already exists, remove it first."
    else
        myapp_dir=$2
        buildImage_dir=$3
        proto_dir=$4

        mkdir $Target_dir

        perl $buildImage_dir/buildImage -r -d $Target_dir \
            $proto_dir/is5.runtime.tar $myapp_dir/myapp.runtime.tar \
            $myapp_dir/myapp.cd.tar $proto_dir/cam.runtime.tar \
            $proto_dir/cmf.runtime.tar $proto_dir/cmfj2.runtime.tar \
            $proto_dir/cmfwd.runtime.tar $proto_dir/db.runtime.tar \
            $proto_dir/dmgt.runtime.tar $proto_dir/eds.runtime.tar \
            $proto_dir/ess.runtime.tar $proto_dir/grid.runtime.tar \
            $proto_dir/jawt.runtime.tar $proto_dir/jchart.runtime.tar \
            $proto_dir/jext.runtime.tar $proto_dir/jgl.runtime.tar \
            $proto_dir/jpwr.runtime.tar $proto_dir/jre2.runtime.tar \
            $proto_dir/jrm.runtime.tar $proto_dir/lotusxsl.runtime.tar \
            $proto_dir/nmcs.runtime.tar $proto_dir/perl.runtime.tar \
            $proto_dir/plug.runtime.tar $proto_dir/pxhlp.runtime.tar \
            $proto_dir/snmp.runtime.tar $proto_dir/svc.runtime.tar\
            $proto_dir/swng.runtime.tar $proto_dir/vorb.runtime.tar \
            $proto_dir/web.runtime.tar $proto_dir/xml4j.runtime.tar \
            $proto_dir/xrts.runtime.tar $proto_dir/eds.cab.tar \
            $proto_dir/jgl.cab.tar $proto_dir/swng2.cab.tar \
            $proto_dir/vorb.cab.tar \
    fi
fi
```

The output of buildImage command is very lengthy. The actual set of tar files may be different from this example. Here is the result:

```
D:\image>ls
<Name of Self-extracting installation file>.exe disk1 temp
autoinstall.sh extract
```

- Now the necessary product (CiscoWorks Home Page, and the customer's application' see the options in ADVANCED_CHOICE section above) can be installed either by running the *Name of Self-extracting installation file*.exe or by running the setup.exe from the d:\image\disk1 directory.

Note To display messages during installation and uninstallation, the myapp.rul file may be prepared and located into myapp\install directory before making the runtime protopackage tar-file.

```
D:\myapp\install>cat myapp.rul

declare
function mypp_preinstall()
begin
MessageBoxLog("Installing MyAP into \n"+NMSROOT,INFORMATION);
end;
function myapp_postinstall()
begin
MessageBoxLog("Running postinstall script for MyAPP",
INFORMATION);
end;
function myapp_uninstall()
begin
MessageBoxLog("Running uninstall script for MyAPP",
INFORMATION);
end;
```

Customizing the Installation Workflow for Windows

The workflow implemented by the CWCS installation framework can be customized for a specific product. You can also modify the workflow to display additional product-specific dialogs at installation time.

Note If you have questions about installation framework customization, contact the CWCS installation team (cmf-install@cisco.com) during the early stages of your project.

The following topics describe how to customize the Windows installation workflow:

- About the Installer Workflow
- Getting Started with Windows Installer Tools
- Creating the Installation Project File
- Creating Install Actions
- Creating Install Panels
- Specifying Conditions For Install Actions and Panels
- Creating the Install Staging Area

- Example: Adding Message Boxes to an Installation
- Example: Creating Custom Password Dialogs
- Example: Adding User Data to Show Details

For information about limited customizations available on Solaris, see the "Customizing the Installation Workflow on Solaris" section on page 21-104.

About the Installer Workflow

The installation framework supplies the installer that:

- Loads the table of contents (disk.toc) and package property files (*.info) from the CD image.
- Performs all steps necessary to interact with the end user during the installation process
- Installs all packages, invoking the package-specific xxx_preinstall and xxx_postinstall functions at the appropriate times.

Some products require that additional dialogs be displayed at installation time. If they are implemented in the xxx_preinstall functions, then the dialogs are displayed too late, and might be hidden in the background. This topic describes how dialogs (or non-interactive actions) can be added to the installation of a specific product.

The installer is implemented as a sequence of steps. Each step is either an action or a panel. The sequence is controlled by the project file. When creating an image, the installation framework build tools load the project file and automatically generate the code that invokes actions and panels and controls the sequence, including the processing of the Next / Back buttons required by the wizard model.

Customizing the installation workflow requires the following steps:

1. Write InstallShield functions implementing the step required for the product installation.
2. Add this step to the project file (see the "Creating the Installation Project File" section on page 21-77).

Getting Started with Windows Installer Tools

To develop InstallShield code that works with the installation framework, be sure your desktop has all required tools. You must have:

- InstallShield Professional 5.5.3 (version 5, maintenance pack 3);
- MKS. In particular, perl, find, and sh utilities must be MKS. You might want to verify the path before trying to build the CD image.

Creating the Installation Project File

The project file is an ASCII file that contains definitions of steps, one step per line. Normally one project file (the template) will contain steps only, with no operations. Other project files will contain additional steps or changes to the template, and use operations with references instructing the generator how to modify the template.

Each step definition uses the following format:

```
[%operation reference][label:]stepName;[sourceFile];condition[;condition[...]]
```

Where:

- `operation` is identified by the % sign. The operation can be any one of the following: `AFTER`, `BEFORE`, `REPLACE`, or `DELETE`.
- `reference` specifies where to add the new step or what step needs to be replaced or deleted. The rest of the line describes the step to be added, replaced, or deleted included.
- If `label` is present, it is separated from other parameters by a colon (:). All other fields are separated by semicolons (;).
- Each step must have a `stepName`. If thestep name begins with `panel_`, then panel-specific code will be generated (see the "Creating Install Panels" section on page 21-79). Otherwise, the step is considered an action (see the "Creating Install Actions" section on page 21-78.
- Following the step is the `sourceFile (`or header) name. The `sourceFile` name must be a file name, not a path name, and can be either the .h or .rul file.
- The `condition` can be either inline or a function call (see the "Specifying Conditions For Install Actions and Panels" section on page 21-82). You can have any number of conditions, but you must have at least one.
- Comment lines are allowed, and must begin with #.

Creating Install Actions

Installer steps are implemented as InstallShield functions. Functions can be either panels or actions. All panels and actions must have a predefined set of parameters, and return codes for the installer to determine the next step. Each panel or action can be accompanied by one or more conditions. The installer evaluates specified conditions and calls the action or panel only if all conditions are positive.

The source code for actions must be included in one of the protopackages in the install/action directory. In the project file, the name of an action is the same as the function, and must be followed by the source file name (without a path). The source file name can be either a header (.h), or a source (.rul) file.

- If a header file is specified, then the installation framework build tools automatically generate the appropriate include statements for the header file, and a matching include for the source file (replacing .h suffix by .rul suffix).
- If a source file is specified, the prototype for the function is generated instead of including a header file.

This gives you two options, as follows:.

Option 1

Put the source code for the custom action into the .rul file with no prototype, and specify the .rul file name in the project. For example:

1. Put the source of myAction into the install\action\myActionSrc.rul file:

```
function myAction()
begin
    ...
end;
```

2. In the project file, add the following:

```
... myAction;myActionSrc.rul
```

Option 2

Put the source code of the custom function into the .rul file and create a matching header file with function prototypes. For example:

1. Put the source of a myAction into install\action\myActionSrc.rul file:

```
function myAction()
begin
    ...
end;
```

2. Put the prototype of a myAction into install\action\myActionSrc.h file:

```
prototype myAction();
```

3. In the project file, add the following:

```
... myAction;myActionSrc.h
```

Normally, Option 1 is more convenient for simple cases, where the action is the only function in the source file. If you want to include multiple functions in a single source file, then Option 2 is the better choice.

Non-Interactive Actions

Non-interactive actions are simple steps that are normally non-interactive, or display simple dialog boxes without Next/Back buttons. Functions implementing such steps must have no parameters, and return codes as follows:

- Return 0 or a positive number to indicate that the step executed successfully and that the installer should proceed to the next step.
- Return a negative number (in the range between -1 and -1000) to indicate that installation must be aborted.

To distinguish actions from panels, the name of non-interactive actions must *not* begin with `panel_`.

Creating Install Panels

Panels are the steps that implement dialogs displayed to the user running the installation. These dialogs use the Next and Back buttons following wizard paradigm. Functions implementing a panel must have:

- A name beginning with panel_.
- One numeric parameter (NUMBER). When a panel function is invoked the value of this parameter is the return code of the previously-displayed panel. Usually, it will be either NEXT or BACK depending on what button the user clicked in the previous dialog.

Note This parameter must be checked if the custom panel displays more than one dialog. If the value parameter is BACK, the step must open the last dialog rather than the first.

The return code of a panel must be one of the following:

- NEXT if user clicked on the Next button and installer should proceed to the next dialog.
- BACK if user clicked on the Back button and installer must return to the step displaying previous dialog.
- SM_SKIP if the panel was skipped. This is important to notify the framework that if user decides to go back from the next dialog this dialog need not to be displayed.
- A negative number in the range between -1 and -1000 to indicate that installation must be aborted.

Note The constants NEXT and BACK are defined by InstallShield. The SM_SKIP constant is defined by installation framework.

The implementation of a panel must be able to handle several special cases:

- In silent mode, the installation must proceed without user interaction. If there is a clear default value for the data requested from the user, that value must be set as if the user entered it. In some cases there are no defaults, and information must be loaded from the answer file (see the "Creating the Answer File" section on page 21-89). The installation framework preloads the name=value pairs from the answer file into the global list lAnswerFile and can be fetched using the getStringFromList function, as shown in Example 21-3.

Example 21-3 Fetching Answers Using getStringFromList

```
if (bQuiet = TRUE) then
    // this is silent mode
    if (getStringPropertyFromList(lAnswerFile,
                    "myParameter", svMyParameter) = 0) then
        // myParameter was specified in answerFile, proceed with this value
        ... // some code that puts svMyParameter into temp location for
            // postinstall, see more on that later
    else
        // value is not specified in the answer file. Your postinstall must
        // be able to handle that situation, if it cannot than the default
        // value must be stored here
        // ...
        // In the case when there can be no default, it is time to abort:
        WriteLogFile("ERROR: the value myParameter is not specified in" +
                          " the answer file, and silent installation can not proceed");
        return -1;     // this instructs installer to abort
    endif;
```

- All dialogs are displayed to the user at the beginning of the installation, and there is always a possibility that the user will decide to move back to an earlier panel or even cancel the installation. Therefore, panel functions must not change anything on the target system. Instead, the data entered by the user should be stored temporarily such that postinstall code can pick up that data and apply it as appropriate. For example:
  - Data can be saved in a temporary file under PROP_DIR. This global variable is populated by installation framework with the pathname of a directory under InstallShield's temporary directory. InstallShield will clean up these files once installation finishes. With the addProperty/getProperty functions, you can use:

    ```
    addProperty(PROP_DIR, "myTempFileName", "myParameter", szValue)
    ```

    in the code of the panel, followed by:

    ```
    getProperty(PROP_DIR, "myTempFileName", "myParameter", svValue)
    ```

    in postinstall.
  - There are two functions that allow you to pass data from panels to postinstall. You can save data using:

    ```
    ListAddString(lTemp, svData1);
    ListAddString(lTemp, svData2);
    ```

```
SetPassEx(lTemp, "myPanelData");
```

In postinstall, you can retrieve data with:

```
if (GetPassEx("myPanelData", svData1, 0) = 0) then
    // use svData1
else
    // that piece of data was not entered by user either because of silent mode
    // or the dialog was skipped
endif;
if (GetPassEx("myPanelData", svData2, 1) = 0) then
    // the same for the second field
    // ...
```

In addition to passing data, these two functions ensure that temporary files are stored in a directory ciphered for the current user, so no other user can view the information. These functions are therefore safe for passwords and other security sensitive data.

- We recommended that dialogs requesting information from the user be displayed with a Custom installation only. In the Typical installation, dialogs should not be displayed. Therefore, panels would normally be used with conditions. The postinstall code trying to apply the data should provide for cases where data was not saved from the dialog, thus allowing you to preserve existing configurations from previous installations or apply the defaults.
- It is important that conditions excluding panels are specified in the project rather than processed internally by the panel function itself. Consider the following implementation of a panel:

```
function panel_myPanel()
begin
    if (...) then
        // this will cause problem, don't do it!!!
        return NEXT;
   endif;
   // displaying actual panel
   return NEXT;
end;
```

In this case, the framework is unaware that the panel was not actually displayed. After calling the `panel_myPanel` function, the framework would proceed to the next dialog. If the user then clicks on the Back button, the framework will invoke `panel_myPanel` again, which is wrong. In order to avoid this situation, a condition must be specified as a function or inline condition in the project. The previous code can be corrected as follows:

```
function panel_myPanel()
begin
    if (...) then
        // this return code notifies the framework that myPanel
        // was not displayed. When user goes back from the next
        // dialog, the framework would jump to the panel that
        // had been displayed before myPanel
        return SM_SKIP;
   endif;
   // displaying actual panel
   return NEXT;
end;
```

Specifying Conditions For Install Actions and Panels

Often, installer steps must be skipped, but the decision to execute or skip a step will be made only at installation time. You can allow for this by specifying conditions for actions or panels. The framework evaluates all conditions before executing an action or panel, and the action or panel function is invoked only if *all* conditions pass. Two types of conditions are supported:

- Condition functions: You can add these just like custom panels or actions. The function does not have any parameters, and must return a positive number if satisfied, or a negative number to skip the action or panel. For the framework to check a condition, the name of the condition function must be specified in the project file. For example:

  ```
  myAction;myAction.rul;needAction;additionalCondition
  ```

  In this example, the action `myAction` will only be invoked if both the *`needAction()`* and *`additionalCondition()`* functions return a positive number.

- Inline conditions: These allow simple checks without creating a function. Build tools simply embed such conditions inside an `if` statement. For example, to run a panel in custom mode, only the panel can be registered in the project file:

  ```
  myAction;myAction.rul;svSetupType="custom"
  ```

Creating the Install Staging Area

Note The installation staging area should be available on a local drive. The following examples assume that the staging area is created in the d:\image directory. If you want the staging area to be in another directory, adjust the commands accordingly.

There are two ways to create the staging area:

- Copy it from the result area of the NMTG automated builds. You will see the image\extract directory under every build. Just copy this directory into d:\image.
- Run the buildImage command. Copy the command from the log file of an automated build and run it locally, changing the destination directory (the value of the -d parameter) to d:\image. This command both creates the staging area and builds the installable image in d:\image\disk1.

After creating the staging area, you can change the install code and rebuild the image as needed using one of the following options:

- Option1:

  ```
  cd d:\image\extract\is5\install
  set debug=1
  sh runme.sh 2>&1 | tee d:\image\log.txt
  ```

  These commands rebuild the CD image from the content of d:/image/extract area. The process takes about 10 to 20 minutes. Because the output of these commands is very lengthy, it is redirected into the file log.txt so it can be analyzed after runme.sh finishes. Setting the debug variable causes runme.sh to skip creating the self-extracting executable (this will save you 5 to 10 minutes), and adds debug information to the image.

- Option 2:

  ```
  set IMAGE=d:\image
  d:/image/extract/is5/install/rul.cmd main projectFiles
  ```

These commands simply rebuild the source code of the installer from the specified project files. The rul.cmd file can be copied into the directory in the path to avoid typing the path each and every time.

The *projectFiles* variable requires that the path names of one or more project files be relative to the d:\image\extract directory. For example, to build the installer based on the CWCS project with additional steps in the file d:\image\extract\kilner\install\action\kilner.project, use the following command:

```
rul main cmf\install\action\core.project
    kilner\install\action\kilner.project
```

This method is available only after buildImage or runme.sh builds a complete image .

Example: Adding Message Boxes to an Installation

To add a simple message box to a CWCS installation:

Step 1 Copy the extract area from the CWCS daily build into the d:\image directory.

Step 2 Build an image in your staging area. To do that, run the following commands:

```
cd d:\image\extract\is5\install
set debug=1
sh runme.sh 2>&1 | tee d:\image\log.txt
```

When runme.sh finishes, the installable image is created at d:\image\disk1. You can run the beginning of installation and cancel it when the Installation Type dialog is displayed. Please note the sequence of dialog boxes before it.

Step 3 Create the function displaying the custom message box. For example, using any text editor, create the file d:\image\extract\cmf\install\action\helloWorld.rul, as follows:

```
// helloWorld.rul - my first custom action
    function helloWorld()
    begin
        MessageBoxLog(“Hello World”, INFORMATION);
                return 0;
    end;
```

Note In this example, additional files are created in the CWCS package area. For actual projects, additional files with custom actions and projects should be added to the packages of the product *using* CWCS rather than CWCS itself.

Note You must create actions and projects under the extract\<pkg>\install\action directories. This way, buildImage or runme.sh can locate such files when the image is built.

Step 4 Add the action to the project. For example, using any text editor, create the file d:\image\extract\cmf\install\action\my.project and add the Hello World rul file to it, as follows:

```
project adding HelloWorld to CWCS installation
    %AFTER panel_welcome helloWorld;helloWorld.rul
```

Step 5 Rebuild the installer. Run the following commands:

```
set IMAGE=d:\image
rul main cmf\install\action\core.project cmf\install\action\my.project
```

Step 6 Now run the installation again. You should see the dialog box with the Hello World message immediately following the Welcome dialog.

Example: Creating Custom Password Dialogs

This example describes how to add a dialog that prompts the user to type a password. It is assumed that CWCS is included in the CD with the product image, and therefore all CiscoWorks dialogs need to be displayed. The dialog described here will be added in such a way that it is displayed by a custom installation only. If the product is installed for the first time, a random password must be generated.

The panel will be implemented by the `panel_myPwd` function. The file with this function will be added to the install\action\mypassword.rul, and the prototype will be added to the install\action\mypassword.h.

Note This example assumes that the installable image of the product is already being built and includes the product protopackages as well as the CWCS protopackages. The image/extract directory must be copied to d:\image directory.

Step 1 Create the installable CD:

```
cd d:\image\extract\is5\install
set debug=1
sh runme.sh 2>&1 | tee d:\image\log.txt
```

Step 2 Add the files mypassword.h and mypassword.rul to one of protopackages (for example, myPkg). There is no need to create new protopackages; just use one of existing product protopackages.

The file extract\myPkg\install\action\mypassword.h contains one line:

```
prototype panel_myPwd(NUMBER);
```

To add the panel to the installer, a new project file is needed (for example, extract\myPkg\install\action\my.project). This project file needs the following line (do not put extra spaces at the start):

```
%BEFORE addUserInputToInfoList
    panel_myPwd;mypassword.h;svSetupType="custom"
```

This condition ensures that the panel is displayed in custom mode only.

The rest of the code goes into the extract\myPkg\install\action\mypassword.rul file. It contains the code for the panel_myPkg function:

```
function panel_myPwd(lastStepRc)
      NUMBER nRc;
      STRING szPwd;
      LIST   lLabels, lTemp;

   begin
      if (bQuiet) then
          // this is quiet mode.
          if (IsInstalled("myPkg") = 0) then
              // this is reinstallation case, leave it to postinstall to handle
              WriteLogFile("INFO: Reinstallation of myPkg, skipping password");
              return NEXT;
          else
              // original installation, let's try answer file
```

```
                if (getStringPropertyFromList(lAnswerFile,
                           "myPassword", szPwd) != 0) then
                    // generate random password
                    randomWord(szPwd, 15, -2, RW_ALL);
                endif;
                // save the password for postinstall to pick up
                lTemp = ListCreate(STRINGLIST);
                ListAddString(lTemp, szPwd, AFTER);
                SetPassEx(lTemp, "myPwd");
                ListDestroy(lTemp);
                return NEXT;
            endif;
        endif;

        // let's do the dialog
        // here you might want to reinitialize the szPwd with the value of
        // existing password in the case of reinstallation / upgrade

        lLabels = ListCreate(STRINGLIST);
        ListAddString(lLabels, "My Password", AFTER);
        nRc = skPassExTitle(
           "Description",
           "text of comment", PASS_DIALOG_ID,"PASS_DIALOG_ID,"", "myPwd",
           PASS_EX_DB_VALIDATION + PASS_EX_EMPTY_ALLOWED + 15,
           lLabels, 0, "Change My Password");
        ListDestroy(lLabels);
        return nRc;
    end;
```

Example: Adding User Data to Show Details

In the CWCS installation, information collected from the user or generated automatically is displayed in the Confirmation Dialog, which allows the user to review the options. This dialog is implemented by panel_installConfirmation. This step generates the standard portion (the one displayed while the details are hidden) and appends the details from the global list lUserOptions. This list is managed by the installation framework; the action addUserInputToInfoList populates CWCS-specific options.

To insert application-specific options into the details, you can either replace addUserInputToInfoList or add another step that appends the application-specific options to the list.

For example, to add the password created by panel_myPwd, create this action:

```
/*-------------------------------------------
 * adds the password to details
 *-------------------------------------------*/
function addMyPwdToInfoList()
STRING svTemp[1024];
begin
    if (GetPassEx("myPwd", svTemp, 0) = 0 && svTemp != "") then
        ListSetIndex(lUserOptions, LISTLAST);
        ListAddString(lUserOptions," My password: " + svTemp, AFTER);
    endif;
end;
```

Note This function must not create the lUserInfoList, but should clean it up.

To add this step to the installation, put the following line into your project file (where the function is in mypassword.rul, and prototype is in mypassword.h):

```
%AFTER addUserInputToInfoList addMyPwdToInfoList;mypassword.h
```

To use application-specific information instead of that created by CWCS, or when your installation does not include CWCS, the code must be modified slightly. For example, to add the password created by panel_myPwd to a non-CWCS installation, or when replacing the details provided by CWCS, create an action that looks like this:

```
/*-----------------------------------------
 * adds the password to details
 *-----------------------------------------*/
function addMyPwdToInfoList()
STRING svTemp[1024];
begin
    EmptyList(lUserOptions);
    if (GetPassEx("myPwd", svTemp, 0) = 0 && svTemp != "") then
        ListSetIndex(lUserOptions, LISTLAST);
        ListAddString(lUserOptions," My password: " + svTemp, AFTER);
    endif;
end;
```

Note This function must not create the lUserInfoList or clean up the list. These tasks are performed by other steps in the installation.

To add this step to a non-CWCS installation, add the following line to your project file (where the function is in mypassword.rul, and prototype is in mypassword.h):

```
%BEFORE panel_installConfirmation addMyPwdToInfoList;mypassword.h
```

To add this step to CWCS installation in which the CWCS details are overridden, add the following line to your project file (where the function is in mypassword.rul, and prototype is in mypassword.h):

```
%REPLACE addUserInputToInfoList addMyPwdToInfoList;mypassword.h
```

Solaris Installation Reference

This topic covers the following reference information for Solaris:

- Setting Ownership for Package Files on Solaris
- Creating the Answer File
- Writing Solaris Scripts
- Using the Solaris Installation APIs
- Using Solaris Build Tools
- Solaris Getting Started Example

Setting Ownership for Package Files on Solaris

The CWCS team has updated installation on Solaris to set the ownership of application files properly during CWCS installation. If you have application files that existed in a previous release and were not part of the installation package, then you must modify the post-install hooks/scripts to add chown/chgrp commands for each file. This includes dynamically created files such as data files and properties files.

The options for setting ownership include:

- Fixed user ownership for all packages in the build
- Fixed user ownership for one package

This section provides information about setting ownership data during the image build process using the buildImage script. The buildImage script expands the protopackages and searches for the runme.sh file and runs it. On Solaris, the .../install/runme.sh comes from the pkgtools protopackage and performs the following tasks:

- It assembles the installable image from expanded protopackages.
- It calls the makesolpkg script to build Solaris packages. During makesolpkg execution for each package, the package content description file called "prototype" is created with file ownership setting as the result of the following procedure.

At the beginning of its execution, the script makesolpkg has the bin:bin ownership hard coded for *$OWNER* and *$GROUP* variables.

Additional facilities have been developed to supply you with the ability to have files owned, as they must be set in the image. These are getting ownership from *package_name*.owner file (for every package). The ownership setting sequence for particular package:

- If the *package_name*.owner file exists, then the ownership is assigned in accordance with the data from this file.
- If there is no *package_name*.owner file, then ownership is assigned as bin:bin from *$OWNER* and *$GROUP* variables.

The prototype file is used by pkgmk utility for package creation.

Related Topics

See the:

- "Setting Ownership from package_name.owner File" section on page 21-87.
- "Setting Ownership During Build/Installation" section on page 21-88.
- "Setting Ownership Assignment Details" section on page 21-88.
- "Understanding and Implementing the casuser" section on page 21-21.

Setting Ownership from *package_name*.owner File

The *package_name*.owner file (see the "Understanding the package_name.owner File" section on page 21-88) is a manifest-like file where you can specify the ownership for files/directories with wild cards. It is necessary to have a separate ownership description file for each package. The rules for this file are:

1. The first string of this file is the header: # This file was automatically created with tocToOwner.pl.
2. The second string contains: DEFAULT_OWNER=*name* DEFAULT_GROUP=*default_owner's_group*

   These two items specify the default owner/group for all the files in the package. This data may be set for any file/directory in the package by explicitly mentioning it in BOM file or by using PROP_IMAGE_OWNER and PROP_IMAGE_GROUP build environment variables.
3. The remaining strings have the structure: *directory/file_name owner_login owner_group*

   These items should be separated by spaces. You can use wild cards for directories (* -s at the end of directory_name string; no any space symbols are allowed between the end of directory name and '*'-symbol). One symbol (*) refers to the current directory only; the ownership described by *owner_login*/*owner_group* is to be assigned for every file/directory in current directory only. Two symbol ** string means recursion; the ownership provided by *owner_login*/*owner_group* is to be assigned for every descendant file/directory starting from those in the current directory.

Understanding the *package_name*.owner File

The protopackages are built as specified by the BOM files. Each package's BOM file provides the ownership data. The most convenient way to set up your .owner file is to use the appropriate .toc file. The .toc file is automatically produced from .bom file during the build procedure and contains more detailed data about your directories/files ownership.

An appropriate script (tocToOwner.pl) was written and the .owner file is now automatically put into protopackage tar-file (along with .bom file and others). So the *packge_name*.owner file is automatically located into right /../install directory.

Setting Ownership During Build/Installation

This topic provides information on typical scenarios for you to set desired ownership during image creating process or during installation. The typical scenarios are:

- Fixed user ownership for all the packages in the build

  To implement this scenario, supply the input parameters OWNER/GROUP in the *pkg_name*.owner file. The makesolpkg file can then get them from DEFAULT_OWNER/DEFAULT_GROUP tags in *pkg_name*.owner file.

  This scenario may be used to set casuser:casuser ownership for all the packages.

- Fixed user ownership for one package

  There are two possibilities for this scenario. First, specify fixed ownership in the .bom file for particular package. Second, prepare the .owner file manually and deliver it to appropriate /install directory during image build.

Setting Ownership Assignment Details

This procedure is implemented by the makesolpkg script. The ownership assignment rule works in the following manner.

- If any particular substring *directory/file name* of string A in *package_name*.owner file matches after applying wildcard rules described above to some directory/file mentioned in the "prototype" file for some package, then the ownership from this string A is assigned to the directory/file in prototype file.
- When there are few possible matches, only the last one from *package_name*.owner file is assigned.
- When no matches are valid for some directory/file in the package, the data from DEFAULT_OWNER/DEFAULT_GROUP items of *package_name*.owner file is used for this directory/file.
- When an absent DEFAULT_OWNER/DEFAULT_GROUP string in *package_name*.owner file (or the absence of this file), the hard coded data is used for ownership assignment (currently it is bin:bin).

The makesolpkg script is located in pkgtools.runtime.tar file. It is untared into the ../pkgtools/install directory. The package is created in two steps:

1. First, the ownership description file prototype is created with pkgproto utility

2. The ownership description file is used as input for pkgmk utility for package build.

After the execution of pkgproto, the prototype file is modified so that the ownership is changed in accordance with ownership assignment rule. Then it creates the package file with the new ownership.

Creating the Answer File

The answer file is an ASCII file that provides the required inputs for quiet installations. It contains the name=value pairs shown in Table 21-20.

Caution For CWCS, the answer file is required because it contains the mandatory `adminPassword` field. If this file is not present for CWCS, setup will exit.

The answer file contains the following name=value pairs:

Table 21-20 Answer File Properties

Property	Description
destination	Optional. Allows quiet installation to install into a directory other than /opt/CSCOpx. If not specified, installation goes into /opt/CSCOpx.
adminPassword	Required for CWCS. For other products, if there are no mandatory fields, the answer file is not required. Specifies the login password for the admin user. It is only used for original installations; reinstallation in quiet mode leaves the password alone.
secretPassword	Specifies the login password for the secret user. It is only used for original installations; reinstallation in quiet mode leaves the password alone.

Notes

- To install a product in quiet mode *when an answer file is not required*, use this command:

```
./setup.sh -q ""
```

In this example, the blank quotes are required.

Example

Answer file:

```
#cat /tmp/answer_file
##Sample Answer file
destination=/opt/cisco
adminPassword=adminpass
```

Setup command:

```
#setup.sh -q /tmp/cscotmp/enpass_cmf
```

Related Topics

For the same procedure on Windows platforms, see the "Preloading the Global List, lAnswerFile" section on page 21-35.

Writing Solaris Scripts

Using the CWCS installation APIs, you can write Bourne shell scripts that will allow you to specify any requirements and enforce any constraints on a package on a Solaris platform.

The following topics describe how to write scripts for your package:

- Using the Solaris Installation Hooks
- Where to Find Solaris Installation Examples

Using the Solaris Installation Hooks

Solaris developers can use the following Bourne shell scripts to perform the required functions for their package installation:

Table 21-21 Solaris Hooks

Hook Type	Description
preinstall	This hook is run by installer before file transfer. Most packages do not need this script. If you need your software to do something (for example, verify prerequisite software) before installing your files, then you need this script.
postinstall	This hook is run after file transfer. It contains the actual configuration of the component. This is the correct place to register daemons.
preremove	This hook is run when the component is uninstalled immediately before removing files. This hook is a good place to unregister daemons or verify that the package is installed.
postremove	This hook is run when the component is uninstalled immediately after removing files. Use this hook to clean up files and directories that are not registered with the packaging system. For example, use this script to remove log directories and files.
prerequisite	This hook is run before any preinstall is performed. It specifies dependencies on other packages, disk space required, and questions to be asked of the installer. If the information obtained by this script is required by another script, such as preinstall, then use AddProperty and GetProperty APIs. If the prerequisite hook is set for the installation to fail, it should return a non-zero return code.

Note Do not ask the user any questions in these files: preinstall; postinstall; preremove and postremove. Instead, define shell variables with reasonable defaults. The installation script will call the package's prerequisite script to ask the questions and set the variables for the other scripts.

If the information obtained by this script is required by another script, such as preinstall, then the script may set an environment variable or write a temporary file. The preferred method is to call AddProperty (see the "AddProperty" section on page 21-98) to save the value and GetProperty (see the "GetProperty" section on page 21-98) in a later script to retrieve the property. These functions are found in commonscript.sh.

Hooks are run in the following sequence:

1. Each prerequisite for all packages.

2. pkgadd for all components. Each pkgadd executes preinstall, file transfer, and postinstall for this package.

At uninstallation time, hooks execute pkgrm for each component. Each pkgrm executes preremove, removes files, postremove.

It is important to understand in which order hooks are executed. The installer observes the following rules at installation:

- The installer skips installation of a component if it decides that a particular component should not be processed. It leaves out a component if its pending version is lower than the installed version. It can leave out the component if it is optional and not required by any other component.
- The installer does not execute a skipped component's prerequisite hook. All other prerequisite hooks are executed before any preinstall is executed.

At uninstallation the installer observes the following rules:

- Shared packages are not removed and their uninstall hooks are not executed unless all installable units they belong to are uninstalled.

Note In general, hooks should not rely on any order of execution. This is especially important in order to provide the same hooks for both major releases as well as for upgrades.

Additional hooks are provided to permit custom operations at the completion of various install operations. For details, see the "Customizing the Installation Workflow on Solaris" section on page 21-104.

Where to Find Solaris Installation Examples

The following installation examples are documented:

- Prerequisite
- Preinstall
- Postinstall

A basic example is provided in the "Solaris Getting Started Example" section on page 21-106.

Using the Solaris Installation APIs

This topic covers the Solaris installation APIs, which you can use in in any hook:

- Using the Solaris Input/Output APIs
- Using the Solaris Package APIs
- Using the Solaris System APIs
- Using the Solaris Installable Unit APIs

Note To use Solaris APIs, add the following line at the beginning of hooks. Remember to use the period at the beginning of the line: `.` *$ {SETUPDIR}*`/commonscript.sh`

The following is a sample function header:

```
###############################################################################
```

```
##   FUNCTION FunctionName <input> [<input>]
## brief description of function.
########################################################################
```

The FUNCTION line shows the function name and all arguments with which it is called.

Note Unless otherwise noted, all functions return 0 for success and 1 for failure.

Using the Solaris Input/Output APIs

The Input/Output APIs handle password management and input/output functions:

- AskPass: Prompts for your password
- AskPassEx: Prompts for your password
- GetPass: Retrieves the password from a temporary file
- ChangeDbPasswd.pl: Changes the Database password in the database and properties file after the installation is completed
- AddPassword: Used at the end of the installation to display the passwords that were either entered or randomly generated.
- PasswordRandomSelection: Returns a randomly-generated password.
- PortRandomSelection: Returns a randomly-generated port number.
- Profile: Returns a randomly-generated port number.
- Debug: Returns a randomly-generated port number.
- PromptResponse: Returns a randomly-generated port number.
- PromptYN: Returns a randomly-generated port number.

AskPass

AskPass *PromptString Temp_File_Name DbPass PassName*

Normally, this API is called in the prerequisite to the installation process. The AskPass API:

1. Prompts you for admin and guest passwords during installation.

   This API asks the password twice to confirm the password.
2. Validates the password, encrypts it and stores the password in a temporary file in the temp directory.

Note This is a shell script function that is available in commonscript.sh.

Arguments

Field	Description
PromptString	Prompt which asks the user for password.
Temp_File_Name	Name of the file where the encrypted password will be stored. This file will be created in the temp directory (/tmp/cscotmp/<temp_filename> directory).

Field	Description
DbPass	Indicates whether to validate the password. When set to **1**: The password is validated according to database rules such as, The password should not start with a number, It should not contain any special characters and password length should not be more than 15 characters. These constraints are imposed by the Sybase DBD. **Note** If the password is for a database, set to **1**. When set to values other than 1: The password is not validated.
PassName	The name of the password. This name is displayed at the end of the installation process using the AddPassword API (see the "AddPassword" section on page 21-96).

Return Values

None

Example

```
AskPass "Enter the CWCS Database Password" "enpass_cmf" 1
    "VALUE_CCSDB_PASSWORD"
```

In this example, the API:

1. Displays the string "Enter the Password" and reads the password. The typed characters will not be displayed on the screen.
2. Prompts the user to retype the password to confirm.
3. If both the password matches and passes the validation, the password will be encrypted and stored in /var/tmp/enpass_cmf file. If the passwords do not match the user will be prompted again for the password. If the password does not pass the validation, an appropriate message is displayed and the user will be prompted again for the password.

AskPassEx

AskPassEx *prompt_string temp_filename pass_Name flag*

This API will be normally called in the prerequisite to the installation process. The AskPassEx API:

1. Prompts users for passwords .
2. Prompts for the password twice, to confirm it.
3. Accepts special characters, such as punctuation marks, ampersands, and so on.
4. Validates the password, encrypts it, and stores it in a temporary file in the temp directory.

Note This is a shell script function that is available in commonscript.sh.

Arguments

Argument	Description
prompt_string	Prompt which asks the user for a password.
temp_filename	Name of the file where the encrypted password will be stored. This file will be created in the temp directory (/tmp/cscotmp/).
pass_Name	Name of the password.
flag	Indicates whether to bypass password validation if the user responds with no entry (`1`) or reprompt for the password to confirm (`0`; this is the default).

Return Values

None.

Example

```
AskPassEx "Enter the CiscoWorks admin password: " "enpass_admin"  "Admin Password" 0
```

In this example, the API:

1. Displays the user prompt string `Enter the CiscoWorks admin password:`.

2. Reads the password the user enters, and stores it in the file enpass_admin. The typed characters are not displayed on the screen.

3. Prompts the user to retype the password to confirm.

4. Assigns “Admin Password” as the name of the password. This name is displayed at the end of the installation process using the GetPass API (see the “GetPass” section on page 21-94).

GetPass

`retVal=`**`GetPass`** *Temp_File_Name*

This API is used to retrieve the password from the temporary file. This API also deletes the password file.

Note This is a shell script function that is available in commonscript.sh.

Arguments

Field	Description
Temp_File_Name	Name of the temp file where the encrypted password is kept. The file should be present in /tmp/cscotmp directory.

Return Values

Field	Description
retVal	0 = success. If successful, the global variable *$PASS* contains the clear text password. 1 = failure

Example

```
retVal=`GetPass enpass_cmf`
```

In this example, the API reads the encrypted password from /tmp/cscotmp/enpass_cmf file, decrypts it and returns to the variable Pass.

ChangeDbPasswd.pl

`$DBSWDIR/conf /`**`ChangeDbPasswd.pl`** *DSN Password*

This API changes the password in the database and in property files. It accepts the DSN name and password as the parameters and changes the password. This function returns 0 if it is successful, and returns a value greater than 0 (> 0) if it has failed.

The API does the following:

- Checks whether the DSN is valid.
- If the DSN is valid, checks whether the database is enabled.
- If the database is enabled, changes the password in the database, odbc.tmpl, .odbc.ini, DBServer.properties and the property file specified in odbc.tmpl.
- If the database is not enabled, changes the password in the database and odbc.tmpl.

This API is normally called in postinstall.

Note This is a Perl function that is available in *$NMSROOT*/objects/db/conf/ directory.

Arguments

Field	Description
DSN	Database name
Password	New password

Return Values

None

Examples

This example changes the password of the CWCS database to “cisco”.

```
$DBSWDIR/conf /ChangeDbPasswd.pl cwcs cisco
```

The following example shows how to change the database password for the CWCS database in the CSCOdb package.

1. `PromptYN "Do you want to Change the CiscoWorks Database Password?" 'n'` . This API asks the user whether the password need to be changed.
2. If the user responds with “yes”, continue as follows:

```
if [${YN} = 'y']; then

        AskPass "Enter the CWCS Database Password" "enpass_cmf" 1

fi
```

3. In the CSCOdb package postinstall:

```
retVal=`GetPass enpass_cmf`

$DBSWDIR/conf /ChangeDbPasswd.pl cmf  $PASS
```

AddPassword

AddPassword "*password_name*" "*password_value*"

This API is used at the end of the installation to display the passwords that were either entered or randomly generated.

Note This utility was introduced in CWCS 2.2 It is a shell script function that is available in commonscript.sh.

Arguments

Field	Description
password_name	Name of the password
password_value	Value assigned to password name (either entered or randomly-generated)

Return Values

None

Example

```
AddPassword "VALUE_CCSDB_PASSWORD" "$PASSWORD"
```

PasswordRandomSelection

```
new_password = "PasswordRandomSelection"
```

This API returns a randomly-generated password. It is used when user interaction is not required to enter the password.

Note This utility was introduced in CWCS 2.2. It is a shell script function that is available in commonscript.sh.

Valid Installation Modes

Typical

Arguments

None

Return Values

A randomly-generated string

Example

```
PASSWORD="PasswordRandomSelection"
```

PortRandomSelection

port_number =”PortRandomSelection”

This API returns a randomly-generated port number. It is used when the designated port is already in use by another process.

Note This utility was introduced in CWCS 2.2 It is a shell script function that is available in commonscript.sh.

Valid Installation Modes

Typical

Arguments

None

Return Values

A randomly-generated port

Example

```
port_no=”PortRandomSelection”
```

Profile

```
Profile label [output file]
```

If in debug mode, write the time stamp for profiling. Concatenate to a file if one is specified.

Debug

```
Debug output string [output file]
```

If in debug mode, write the string. Concatenate to a file if one is specified.

PromptResponse

```
PromptResponse prompt string default string
```

Prompts user for input and store response in variable RESPONSE.

PromptYN

```
PromptYN prompt string default string
```

Prompts user for a yes or no respones and stores the response in lower case in the variable YN.

Using the Solaris Package APIs

This group of APIs contains package functions:

- AddProperty
- GetProperty

- UpdateAnsFile
- RunRequestScript
- SetInstallMode_SOL
- UpdateInvFile
- Installf
- CheckPkgInstalled
- GetPkgParam
- needsMoreDiskSpace
- SetInstallPkgMode_SOL
- SetInstPkgMode_frmPrePst

AddProperty

```
AddProperty package name property name property value
```

Adds a property to a package.

GetProperty

```
GetProperty package name property name
```

Retrieves a package's property and stores its value in variable PROPVALUE.

UpdateAnsFile

```
UpdateAnsFile package name
```

Calls this function at the beginning of postinstall.

RunRequestScript

Obsolete.

SetInstallMode_SOL

Obsolete.

UpdateInvFile

```
UpdateInvFile fileset path permissions uid gid
```

Updates the Fileset.inventory file. This function is necessary if you need to modify the attributes of files or directories that are part of the distribution.

Arguments

Field	Description
fileset	Fileset being processed
path	Installer to be modified. This is used as a key to locate a record.

Field	Description
permissions	Permissions.
uid	Owner.
gid	Group.

Installf

This is a platform-independent wrapper providing the functionality of the Solaris install function (installf). For parameters, refer to the Solaris man page on installf.

Use this inside pre- or post- installs if you are modifying or updating data in your package and must ensure that your changes are valid.

CheckPkgInstalled

`CheckPkgInstalled` *`packagename`*
Verifies the existence of a package.

GetPkgParam

`GetPkgParam` *`packagename parametername`*

Gets the value of the parameter for the package. This is an abstraction of pkgparam -v pkg param_name for Solaris.

needsMoreDiskSpace

`needsMoreDiskSpace` *`packagename`*

Gets the numeric value of the dynamically determined disk space required for installation and the estimated size required for use in MB from *pkg*.nmds.

SetInstallPkgMode_SOL

`SetInstallPkgMode_SOL` *`package_name`*

This API sets the package installation mode after checking if that the package already exists.

Note Use this utility only in the prerequisite to the installation process.

Note This utility was introduced in CWCS 2.2 It is a shell script function that is available in commonscript.sh.

Arguments

Field	Description
package_name	The name of the package

Return Values

Field	Description
PKG_I_MODE	Installation mode options: • NEW • REINSTALL • UPGRADE

Example

```
SetInstallPkgMode_SOL CSCOapch
```

SetInstPkgMode_frmPrePst

```
SetInstPkgMode_frmPrePst packagename
```

This API sets the package installation mode after checking if the specified package already exists.

Note Use this utility only in preinstall and postinstall.

Note This utility was introduced in CWCS 2.2 It is a shell script function that is available in commonscript.sh.

Arguments

Field	Description
packagename	The name of the package

Return Values

Field	Description
PKG_I_MODE	Installation mode options: • NEW • REINSTALL • UPGRADE

Example

```
SetInstPkgMode_frmPrePst CSCOapch
```

Using the Solaris System APIs

This group of APIs includes system functions:

- GetBootScript

- GetInitDir
- GetLibPath
- NetstatForPort
- PortUsed
- SaveBackFile
- RestoreBackFile
- RemoveBackFile
- DelServices
- AddServices
- GetDF
- GetFreeDF
- MakeDir
- GetOS

GetBootScript

Retrieves a package's boot script.

GetInitDir

Retrieves a package's initial directory.

GetLibPath

Retrieves the library path.

NetstatForPort

```
NetstatForPort portnumber
```

Verifies that `portnumber` is used. Returns 1 if port is in use, else 0.

PortUsed

```
PortUsed portnumber tcp or udp
```

Verifies that a port is used. Return 1 if port in use, else 0.

SaveBackFile

```
SaveBackFile file name
```

Saves a copy of file. Use before editing the file.

RestoreBackFile

```
RestoreBackFile file name
```

Restores a copy of file. Use when editing has failed.

RemoveBackFile

```
RemoveBackFile file name
```

Removes a copy of this file.

DelServices

```
DelServices service name
```

Removes an entry from /etc/services.

AddServices

```
AddServices service name service port service protocol [comment]
```

Adds an entry to /etc/services.

GetDF

```
GetDF file system
```

Raw output from df or bdf command.

GetFreeDF

```
GetFreeDF file system
```

Gets the numeric value of the disk space for this file system.

Note GetFreeDF expects you to specify the name of the file system , but does not check that you have done so. If you fail to specify a file system, it will get the numeric value of the disk space for the file system last returned by the df -k command.

MakeDir

```
MakeDir directory name
```

Ensures that the proposed directory exists. If not, creates the directory and sets the owner and group to **casuser**. If the directory exists, MakeDir assumes that the owner and group are **casuser**.

GetOS

Sets the environment variables LC_OS and OS to our canonical form of the OS name.

Using the Solaris Installable Unit APIs

These APIs include the installable unit functions:

- GetNMSRoot
- SetIMode
- GetIMode

GetNMSRoot

Retrieves the previously stored value of NMSROOT.

SetIMode

```
SetIMode install mode
```

Stores the value of NMSROOT.

GetIMode

Stores the value of I_MODE.

Using Solaris Build Tools

This topic provides instructions to build an image from protopackages using the installation framework on Solaris. The main steps are:

- Step 1: Install Third-Party Tools On Solaris
- Step 2: Install the Framework On Solaris Platforms
- Step 3: Prepare the Make Image on Solaris

The following topics provide additional guidelines and examples:

- Customizing the Installation Workflow on Solaris
- Debugging on Solaris
- Verifying Packages on Solaris
- Solaris Getting Started Example

Step 1: Install Third-Party Tools On Solaris

Install the Solaris-specific third-party tools referenced in the "Including Files in the Protopackage" section on page 21-20.

Step 2: Install the Framework On Solaris Platforms

To install the framework, copy the following files to the Solaris working directory:

- buildImage: Main script which creates the CD image from protopackages
- verifyImage: Script that verifies the structure of the protopackages.
- is5.runtime.tar: Protopackage containing the Windows installation framework.
- pkgtools.runtime.tar: Protopackage containing the Solaris installation framework.
- makesolpkg: Solaris -only script

Step 3: Prepare the Make Image on Solaris

Follow these steps to make the image on Solaris:

Step 1 Verify that MKS is in your path.

```
which sh
which perl
perl -v
```

The output displayed for these commands should show valid paths for each tool. Perl needs to be version 5.5 or higher.

Step 2 Run buildImage command:

```
perl buildImage -r -d image_path toolpath/pkgtools.runtime.tar rtpath/myPkg.runtime.tar
protopackages
```

Where:

- `-r` is the option needed to refresh the `image_path` (by first removing all directories from `image_path/extract` and then extracting all protopackages).
- `image_path` is the full path name of the directory where the image will be created. Normally, free space of at least three times the size of runtime will be required. You must specify the path starting from the top level directory, with forward slashes, no spaces allowed.
- `toolpath` is the full path to the pkgtools.runtime.tar file.
- `rtpath` is the full path to your runtime.tar file.
- `protopackages` is a space-delimited list of all protopackages to be installed. Each protopackage specification mustinclude full path names with forward slashes, no spaces allowed.

This command will create the following directories under `image_path`:

- extract subdirectory: Contains all files extracted from protopackages.
- disk1 directory: Contains the installable image.

Customizing the Installation Workflow on Solaris

The CWCS installation framework workflow on Solaris does not offer the extensive custom action and panel features available on Windows (for details on these features, see "Customizing the Installation Workflow for Windows" section on page 21-76) .

However, the Solaris framework does offer two features that allow you to customize most installation workflows with a great deal of flexibility, as explained in the following topics:

- Collecting User Interactions at the Start of a Solaris Install
- Executing Custom Operations During a Solaris Install

Collecting User Interactions at the Start of a Solaris Install

You can streamline the user's workflow and make possible "unattended" Solaris installs by prompting the user for all installation inputs at the start of the install, instead of during each install phase when they are required. To do so, simply add the following tag to the corresponding info file:

```
USER_INTERACTION=TRUE
```

You may want to change the wording of the install prompts to ensure that they make sense to users when presented in a single group.

Executing Custom Operations During a Solaris Install

To execute custom operations at various steps during the install, using the hooks shown in Table 21-22.

Table 21-22 Solaris Workflow Customization Hooks

Hook	Executes Your Script
AFTER_LICENSE	After license display
AFTER_INSTALL_TYPE	After the user selects the install type
AFTER_PREREQS	After running prerequisites
AFTER_PKG_ADDS	After package additions

You can specify conditions, titles and scripts for these operations much as you do for action customizations on Windows. For example, you can add the following section to your dsk.toc file to execute the script rmJrunWeb.sh after performing a prerequisite test to ensure that a version of CWCS earlier than 3.0 is present:

```
[AFTER_PREREQS]
AFTER_PREREQS_0=UNINSTALL_JRE2_JRUN_WEB

[UNINSTALL_JRE2_JRUN_WEB]
AFTER_PREREQS_0_CONDITION=cmfwd.2.1.0-2.2.9
AFTER_PREREQS_0_title=Uninstall CSCOjrun CSCOweb CSCOjext CSCOjre2
AFTER_PREREQS_0_script=rmJrunWeb.sh
```

In this example, the script will execute only after prerequisites are checked, and only if the condition is met. The title will be displayed before the script is executed.

Debugging on Solaris

To debug on Solaris, use generic shells. No additional coding is needed. Use set -x to add generic debugging information to your log file.

Verifying Packages on Solaris

The Solaris version of the CWCS installation framework provides the pkgchk command to permit package integrity verification. You can use pkgchk to perform package verification:

- Before installation, within the image.
- After installation.

To verify a package before installation, run pkgchk as follows:

```
pkgchk -d location name
```

Where:

- `location` is the package location within the image.
- `name` is the package name.

For example :

```
pkgchk -d  disk1/packages CSCOapch
```

If a pre-installation verification fails, it will report the following errors:

```
ERROR: The following base package image is bad: CSCOname
```

```
ERROR: Package verification failed : CSCOname aborting.
```

If you get this kind of error, you can be sure that there was a problem during package creation in the build. Once the installation has aborted, you can try the command on the command line, to find out why it failed

To verify a package after installation, run pkgchk as follows:

```
pkgchk -n name
```

Where `name` is the package name.

Package verification may fail after installation if the files:

a. Were modified during postinstall. In this case, you may have to use the installf command during postinstall (for details, see the "Installf" section on page 21-99) to fix the problem.

b. The files are volatile (that is, they change in size, as with configuration files). In this case, you may need to use installif to declare this file as volatile.

Note When using installf, be sure not to use the NMSROOT variable in the file path. For example, if the path is /opt/CSCOpx/examples/example1, give this path explictly, not as $NMSROOT/examples/example1. If you use NMSROOT, you will experience problems during custom path installation.

Solaris Getting Started Example

The following is an example of how to create an installable CD image of CWCS for a customer application. The *myapp* name refers to the application *my application*; it is truncated because Solaris package names must have five letters or less on Solaris (CSCOxxxx).

Before you use this example, verify that the tools are in the path.

```
# which sh
```

The return value should display

```
/bin/sh
```

Perl V5.0 or higher is required.

```
# which perl
```

The return value should display

/auto/em_tools/sol/bin/perl

```
# perl -v
```

The return value should display

```
This is perl, version 5.005_03 built for sun4-solaris
```

```
Copyright 1987-1999, Larry Wall
```

For this example, it is assumed that:

- The sample application is in two tar-files, myapp.cd.tar and myapp.runtime.tar, both in the current directory. The installation script untarrs these files automatically and places them in the following target directories: myapp.cd.tar into myapp\disk1 directory structure and myapp.runtime.tar into myapp\install and myapp\runtime structures. The directory myapp\disk1 contains the disk.toc file. The following disk.toc describes our sample application's table of contents.

```
[RELEASE]
NAME=CWCS with Test Application
VERSTR=1.0

[COMPONENTS]
TAGS=cwcs cmfwd cmfj2 CSCOmyapp
UNINSTALLABLE=cmfwd CSCOmyapp
VISIBLE=cwcs cmfwd CSCOmyapp
CHOICE=cwcs cmfwd CSCOmyapp
DEFAULT=ALL

[ADVANCED_CHOICE_1]
ADVANCED_CHOICE_1_CONDITION=cmfwd.1.0.0-1.9.99
ADVANCED_CHOICE_1_TYPE=EXCLUSIVE
ADVANCED_CHOICE_1_DEFAULT=4
ADVANCED_CHOICE_1_1_TEXT=CiscoWorks Common Services (CWCS) Base Desktop
ADVANCED_CHOICE_1_1_TAGS=cmfwd
ADVANCED_CHOICE_1_2_TEXT=CWCS (including Base Desktop)
ADVANCED_CHOICE_1_2_TAGS=cwcs
ADVANCED_CHOICE_1_3_TEXT=myapp Application
ADVANCED_CHOICE_1_3_TAGS=CSCOmyapp
ADVANCED_CHOICE_1_4_TEXT=myapp Application and CWCS
ADVANCED_CHOICE_1_4_TAGS=cwcs cmfwd CSCOmyapp

[ADVANCED_CHOICE_2]
ADVANCED_CHOICE_2_CONDITION=TRUE
ADVANCED_CHOICE_2_TYPE=EXCLUSIVE
ADVANCED_CHOICE_2_DEFAULT=3
ADVANCED_CHOICE_2_1_TEXT=CiscoWorks Common Services (CWCS) Base Desktop
ADVANCED_CHOICE_2_1_TAGS=cmfwd
ADVANCED_CHOICE_2_2_TEXT=CWCS (including Base Desktop)
ADVANCED_CHOICE_2_2_TAGS=cwcs
ADVANCED_CHOICE_2_3_TEXT=myapp Application and CWCS
ADVANCED_CHOICE_2_3_TAGS=cwcs cmfwd CSCOmyapp
```

For this toc file you need the following:

- REGISTRY_ROOT provides the key name for component information. The value has been set for the CWCS release and has to be the same to keep compatibility with CWCS.

Note The NAME parameter in the [RELEASE] section is used to make a name of an executable. All spaces are replaced by underscores, but parenthesis, comma slashes, and other characters are not allowed in filenames.

The ADVANCED_CHOICE section determines what is the correct scenario depending on what is already installed on the target machine. In the case where the target machine has a previous version of CiscoWorks Common Services (CWCS) Base Desktop already installed, the customer has the following options:

- Upgrade to a new version of CiscoWorks Base Desktop
- Install CWCS 3.0
- Install the myapp application on top of CiscoWorks Desktop, or

  - Install the application bundled with CMF 3.0.

  In the case of a fresh installation, you do not have the option of a separate myapp installation. In this case, there are only the remaining three options mentioned in the ADVANCED_CHOICE_2 section.

- The directory myapp/install contains the myapp.bprops and mypp.pkgpr files. The directory myapp/runtime contains the subdirectories "cgi-bin" and "htdocs", corresponding to the structure of similar directories of CWCS to be used by the web server. Substitute the two Perl scripts, cgi-bin/myapp/myappcgi.pl and htdocs/myapp/myappframe.html, with your application files. The following files are for linking our HTML and Perl files to the appropriate tree structure of the CWCS main web page:
  - htdocs\Xml\System\maintree\myapp.xml
  - htdocs\Xml\System\maintree\myappcgi.xml

  Each developer must swap their XML files in their place. For details on integrating your application with CiscoWorks, refer to the "Integrating Your Application with CWHP" section on page 7-6.
- pkgtools.runtime.tar file is in protopackages directory of the SDK kit.
- BuildImage file is in the root directory of the SDK kit.
- makesolpkg file is in the root directory of the SDK kit.

To create a CWCS image, use the sample1.sh in the current directory.

- Launch sample1.sh with four arguments:

  ./sample1.sh *arg1 arg2 arg3 arg4*

  where:
  - *arg1* is the target directory;
  - *arg2* is the directory with files myapp.cd.tar and myapp.runtime.tar (actually current directory)
  - *arg3* is the directory with buildImage file
  - *arg4* is the directory with pkgtools.runtime.tar file, protopackages tar files, cab-files.

  The sample1 file is shown below:

```
#!/bin/sh
########################################################################
# INPUT:
# 1 - Target_dir
# 2 - myapp_dir
# 3 - buildImage_dir
# 4 - proto_dir pkgtools.runtime.tar and protopackages dir
# you should use to execute this script
########################################################################

if [$# -ne 4]; then
        echo "ERROR:sample2.sh called with insufficient args."
else
        Target_dir=$1
        myapp_dir=$2
        buildImage_dir=$3
        proto_dir=$4

        rm -rf $Target_dir
        mkdir $Target_dir

        /usr/local/bin/perl $buildImage_dir/buildImage -r -d $Target_dir \
        $proto_dir/pkgtools.runtime.tar \
    $myapp_dir/myapp.runtime.tar $myapp_dir/myapp.cd.tar \
    $proto_dir/cam.runtime.tar $proto_dir/cmf.runtime.tar \
    $proto_dir/cmfj2.runtime.tar $proto_dir/cmfwd.runtime.tar\
    $proto_dir/db.runtime.tar $proto_dir/dmgt.runtime.tar \
    $proto_dir/eds.runtime.tar $proto_dir/ess.runtime.tar \
    $proto_dir/grid.runtime.tar $proto_dir/jawt.runtime.tar \
    $proto_dir/jchart.runtime.tar $proto_dir/jext.runtime.tar \
    $proto_dir/jgl.runtime.tar $proto_dir/jpwr.runtime.tar \
    $proto_dir/jre2.runtime.tar $proto_dir/jrm.runtime.tar \
    $proto_dir/logmsg.runtime.tar $proto_dir/lotusxsl.runtime.tar \
    $proto_dir/nmcs.runtime.tar $proto_dir/perl.runtime.tar \
    $proto_dir/plug.runtime.tar $proto_dir/pxhlp.runtime.tar \
    $proto_dir/snmp.runtime.tar $proto_dir/swng2.runtime.tar \
    $proto_dir/jext.runtime.tar $proto_dir/vorb.runtime.tar \
    $proto_dir/web.runtime.tar $proto_dir/xml4j.runtime.tar \
    $proto_dir/xrts.runtime.tar $proto_dir/eds.cab.tar
    $proto_dir/jgl.cab.tar $proto_dir/swng2.cab.tar
    $proto_dir/vorb.cab.tar

fi
```

The output of this command is very lengthy. The actual set of tar files may be different from this example. Here is the result:

```
#cd <Target Directory>
#ls
autoinstall.sh disk1 extract
```

- Now the necessary product (CiscoWorks Base Desktop, CiscoWorks, customer's application see the options in ADVANCED_CHOICE section above) can be installed by running setup.sh from the ./disk1 directory.