March 2001
DTN-14, Revision 0-0

Creating Package Files for the Qube 3

1 Overview

New software, updates, and third-party software for Cobalt server appliances is delivered in the form of downloadable software *.pkg files, referred to as *package files*. Package files are available from Cobalt Network's web and FTP sites, and from third-party vendors. Cobalt uses package files to distribute and install software because it fits the Cobalt philosophy of ease of use: users can install software using only a web browser, which is easier and more intuitive than telnet or other Unix command line interfaces.

This note describes the typical contents of a package file, and how to create one for the Qube 3" . For more complete information on all aspects of developing applications for the Qube 3, go to `ftp://ftp.cobaltnet.com/techinfo/technotes/SSDK.pdf`.

Table of Contents

1.1 Audience

The audience for this technical note are developers of software for the Sun Cobalt Qube 3. For more information, see `http://developer.cobalt.com`

2 What is a Package File

A package file is a single downloadable compressed collection of files used for software installation or updates for Cobalt Network's Qube 3.

To create an application, you must create a module that includes all the components needed and structure it so that it can be easily installed by users, in a package file format (`.pkg`). This technical note lists the fields that you need to include so that the Qube 3 can display the appropriate information during the installation process. It also describes the appropriate directories, files, and resources for your application module.

2.1 BlueLinQ

BlueLinQ " is the software notification system for Cobalt's Qube 3 and RaQ XTR products. BlueLinQ is a Cobalt innovation that gives you instant access to product updates and new services as they become available.

Using Cobalt's BlueLinQ technology, your Qube 3 informs you when new software is available. With a click of a button, you can download the new software package and automatically install it.

3 About the Application Module

The application module is a self-contained bundle of files, directories, and resources required for a new capability. Depending on the type of module you are creating, you choose the appropriate level of integration. Some modules trigger both the user interface and the back end system; others are stand alone modules.

New modules can contain any or all of the following code:

1) User Interface (UI) modules

- UI pages built using UIFC
- Navigation nodes, such as adding buttons and menu items

The Web mail service that is displayed on the Cobalt menu is an example of a service that is integrated only with the user interface and uses IMAP as its back-end system. The files for the user interface go into the `ui` directory; for more information about module directory layout, see <f_Link> on page –4.

1) Internationalization modules

- Internationalization resources to translate the user interface into other languages.

2) Back-end modules

- CCE configuration files
- CCE handlers

Adding a user to the Qube 3 is an example of an instance that impacts only the back-end modules, where the existing user interface is used and the CCE configuration files and handlers are invoked.

3) Binary modules

- Service binary and configuration files, for example, email modules have `SendMail` and `Majordomo` binaries and modify the configuration files for these binaries.
- Databases that register users as they are created and notify event handlers about creating users. This type of module uses the existing user interface.

These modules can be manually installed and completely unintegrated to the Cobalt User Interface (UI).

4 Naming Your Application Module

Developers must use unique vendor-specific names for modules to avoid name conflicts.

Note
Cobalt uses *base* in its module names, for example, `base-devel.mod.` Developers **must** differentiate their modules by naming the modules with a distinctive name, preferably a name that reflects their company or product, for example, *vendor_name*_module.

5 Building a New Service Module

A service module is a self-contained bundle of files or directories and resources required for a new capability, for example, an ecommerce product or a system backup product. New modules can contain any or all of the following:

- Navigation nodes — `service.xml`
- User Interface (UI) pages built using UIFC — `service.php`
- Internationalization resources — `service.po`
- CCE configuration files — `service.schema`, `service.conf`
- CCE handlers — `serviceMod.pl`, `serviceMod.c`
- Service binaries and configuration — `serviced`

Note
You can write handlers in any language. Cobalt provides bindings for C and Perl.

Cobalt enabling tools include:

- Standard directory structure document; see Figure 5–1, "Module File Hierarchy," on page 15.
- Build tools to create loadable module files (scripts and a Makefile)

6 Making your Application into a Package

This section describes the skeleton module for Sausalito. By customizing the skeleton module for your needs, you can integrate seamlessly into the Cobalt configuration environment.

To build a service module:

1) Create handlers to interact with the Cobalt Configuration Engine (CCE). A configuration file goes in `glue/conf`; the actual handlers go in `glue/handlers`.
2) Create any user interface components, if necessary. These include web and menu page descriptors, which go in the `ui/web` and `ui/menu` directories, respectively.
3) Write any `locale` files; these go in the `locale` directory.
4) Look at `templates/spec.tmpl` and `templates/packing_list.tmpl`.

Note
The default template to build RPM files is in `/usr/sausalito/devel/templates`. If you want to modify these templates, create a template directory in your module. Copy these files to your template directory and modify them as needed.

---

5) Look at the top-level `Makefile`. Adjust the variables to fit your situation.

The default build targets are `make all`, `make clean`, `make install`, and `make rpm`.

Note
A sample skeleton module is available in the Cobalt Developer web page. Go to `http://developer.cobalt.com/devnet/devtools.html` for the code sample and Readme file.

Here's some more information about the default `make` rules and expected file names:

Table 1: The top-level `Makefile` variables

Makefile Variables	Description
VENDOR	the vendor field for your module
VENDORNAME	the actual vendor name; this name can be the same as VENDOR
SERVICE	the name for the service
VERSION	version number
RELEASE	release number
BUILDARCH	set to `noarch` if you don't want platform-specific RPMs generated.
XLOCALEPAT	set to a space-separated list of locale patterns to exclude
BUILDUI	packages all files in `ui/web` and `ui/menu`.
BUILDLOCALE	set to `yes` to build these components, create RPMs, and create a capstone RPM.
BUILDSRC	build the files is in the `src` directory.
BUILDGLUE	If `BUILDGLUE` is set to `yes`, packages all the handlers, object schemas, configuration files for event triggers, and conf files. If set to `no`, `BUILDGLUE` does no packaging.
DEFLOCALE	This locale is used for static HTML pages, for example, `en` or `ja`.

The BUILD variables determine which directories to include when calling the `clean`, `install`, and `rpm` targets.

The default `make` rules take the `BUILD?` variables and build up `ui`, `glue`, and `locale` RPMS, if requested. If any of these RPMS are generated, a *capstone* RPM is created as well. A capstone is a type of packing list for the RPMs.

Table 2: Module Directory Layout

Directories	Description
constructor	capstone constructors
destructor	capstone destructors
glue	handler and configuration modification code
ui	user interface and user interface menu code
locale	locale information and locale-specific UI pages

Table 2: Module Directory Layout

Directories	Description
templates	user-modifiable template files used in packing list and RPM generation
src	`src` directory (optional)
RPMS	RPMS directory (optional)
SRPMS	source RPMS directory (optional)

The default `make` rules expect the following file layout:

1) `glue/conf/*`
   `glue/handlers/*`
2) `locale/localeX/$(SERVICE).po`
3) `ui/menu/*`
   `ui/web/*`
4) `constructor/*`
   `destructor/*`

The default `make` rules place these files in the following locations:

```
glue/conf/* -> $(CCEDIR)/conf/$(VENDOR)/$(SERVICE)/*
      glue/handlers/* -> $(CCEDIR)/handlers/$(VENDOR)/$(SERVICE)/*

locale/localeX/$(SERVICE).po ->
      /usr/share/locale/localeX/LC_MESSAGES/$(VENDOR)-$(SERVICE).mo

ui/menu/* -> $(CCEDIR)/ui/menu/$(VENDOR)/$(SERVICE)/*
      ui/web/* -> $(CCEDIR)/ui/web/$(VENDOR)/$(SERVICE)/*

constructors/* $(CCEDIR)/constructor/$(VENDOR)/$(SERVICE)/
      *destructors/* $(CCEDIR)/destructor/$(VENDOR)/$(SERVICE)/*
```

If your module does not contain compiled code, the above `make` rules should be all that you need for building a service module. Otherwise, you need to know about a couple additional `make` rules. First, `make` checks for Makefiles in the `glue`, `src`, and `ui` directories and uses them, if they are present. You must prepend the `PREFIX` environment variable on the install phase of the Makefile so that RPMs are properly generated.

In addition, the `make rpm` rule does not touch the `src` directory, so you must create any RPMs you want from separate specification files. `templates/packing_list.tmpl` should be updated to reflect any of these RPMs without version numbers. You should create a file with the same name as the RPM in the `rpms` directory to get the appropriate version included in the packing list.

Finally, you might need to edit `templates/rpmdefs.tmpl` to add additional build, install, and file targets for any files that you have. The `<begin [$%]VARIABLE>` sections in the `rpmdefs.tmpl` file correspond to the same `[VARIABLE_SECTION]` sections in `templates/spec.tmpl`. If you want to add something to `spec.tmpl` that is not dependent upon a single RPM, you can directly add it to `spec.tmpl`.

Note
If you have a VENDORNAME specified, `make` searches first in `{glue, locale, ui, src}/$(VENDORNAME)` for files before searching in the `glue, locale, ui,` and `src` directories.

7 How to Install your Package File on the Qube 3

There are two ways that packages can be installed on Qube:

- manually
- update server

Both these ways provide information about the package, that is, package meta-information, before the user installs the package. This meta-information includes fields with the package name, vendor, description, license, and whether package dependencies exists; these fields are described in Table 3. This information is needed to properly display in the Qube UI details about the package before its installed. To provide this information, this information is included in the package list and the package information directories for each package.

Update servers alert you if they have new software for your Qube 3. When the Qube is alerted that there is a new version of software for the Qube, the update server and Qube have the following dialog:

1) The Qube 3 queries the server for information about new software. It provides details about the Qube including the packages installs, Qube identification, and so forth.
2) The update server replies with list of available packages with associated information, such as license and locale information. This informations corresponds to the `packing_list` and the contents of the `pkginfo` directory.
3) If an `InfoURL` field is specified, a popup window with the URL is displayed when you go to the install detail page. If an `InfoURL` field is not specified, a short description of the package is displayed.
4) Installation can be selected.

The events around the manual installation are as follows:

1) The user on the Qube enters the package location through either browser upload, URL download, or putting the file in `/home/packages.`
2) The Qube prepares the package for installation and displays the installation page. This informations corresponds to the `packing_list` and the contents of the `pkginfo` directory.
3) The contents of the installation page display a short description of the package that is to be installed.
4) Installation can be selected.

8 Installation Process

The following stages occur in the installation process:

- If the package requires the server to reboot, the user is prompted to reboot the machine.

- The install process looks first for a `splash` page If the `splash` page specifies the `pre-installation` option, it will look for an `index.cgi` or `index.php` page to call. It will pass in the following two variables a GET request to these files: `submiturl` and `cancelurl`.

Note
The `splash` page optionally specifies a pre-installation page, which allows developer to create a custom page for the package including license information. This page must be a CGI or PHP page that accepts `GET` requests.

- If the splash page doesn't exist and the license field does, **BlueLinQ** will present a standard license page containing the value of the license field.

Note
The Qube 3 software notification mechanism is called **BlueLinQ**.

- Once the user accepts the license (if there is a license), **BlueLinQ** checks package dependencies, and halts if there is a dependency error. If not, **BlueLinQ** runs the pre-installation scripts, install RPMS, and then runs the post-installation script. The scripts are located in the `scripts` directory of the package.

Note
BlueLinQ will install an RPM only if it is newer than any existing RPMs. If there is an existing RPM on the server, **BlueLinQ** increments the reference count each time you add a package with a RPM referenced in it. When you uninstall a package, the reference count is reduced. If the reference count for a package is less than one, **BlueLinQ** deletes the RPM.

8.1 Choices for the Installation Process

You can customize your installation. You can change the look and feel of install by opting to include:

- an `infoURL` field
- a splash page
- a generic license

The splash page must be a CGI or PHP file. The update process calls this CGI with the following URL variables set: `submitURL` and `cancelURL`.

9 Package Structure

The package file format is a `tar.gz` file. When you install a package file, **BlueLinQ** checks for the following items:

- whether the file is a `tar` file or a compressed `tar` file
- whether the file is signed

In packages for earlier Cobalt products, package files had the following elements:

- packing_list

- RPMs
- SRPMs
- `install_me` script

Packages for earlier Cobalt products had scripts that performed all installation tasks. Package dependency checking was done by the package itself. New packages have scripts that runs at specified times.The scripts deal with the following issues:

- pre-installation
- post-installation
- pre-uninstallation
- post-uninstallation

BlueLinQ runs these scripts as part of the installation. Package dependencies are based on vendor name, version number and package name. You can evaluate version number to determine if they are equal, less than, or greater than the target version. Sausalito currently checks a three-part field, for example, 1.0 or 1.1.2.

The new packing list format includes the following elements as shown in Table 3, "Package List Format," on page 8.

Note
All the information in the package list format is case-sensitive.

Table 3: Package List Format

Component	Description
[Package -- Version=1.0]	
`Vendor`	vendor name can include alphabetical characters, numbers, underscore (_), and the plus sign (+). Spaces and hyphens (-) are not permitted.
`VendorTag`	internationalizable vendor string
`Name`	*packagename* can include alphabetical characters, numbers, underscore (_), and the plus sign (+). Spaces and hyphens (-) are not permitted.
`NameTag:`	internationalizable package name string.
`Category`	category information can include alphabetical characters, numbers, underscore (_), and the plus sign (+). Spaces and hyphens (-) are not permitted.
`Location`	URL that specifies the package download location
`InfoURL`	additional information URL. Optional. Use this if you want to display a new site (as opposed to installing a package).
`InfoURL` options	options that should be sent with to the URL, which can include serial number, product identifier (product), and vendor name (name).
`Version`	version of the package

Table 3: Package List Format

Component	Description
`Version Tag`	Internationalizable version number.
`Size`	size in bytes (only used by the update server.)
`Product:` NOTE: use this field to specify as many products as you are including. Include one line for each package. You can use a regular expression to specify products, for example: (4000|4010|4100) WG.	Cobalt product requirements: for example, 4100WG or 4*nnn*WG. NOTE: 4000WG is the product number for the basic Qube 3. 4010WG is the product number for the Qube 3 with caching; 4100WG is the product number for the Qube 3 with caching and mirroring.
`PackageType`	specify `complete` or `update`
`Options`	`uninstallable, reboot, refreshui, refreshcce`
`LongDesc`	internationalizable long description
`ShortDesc`	internationalizable short description
`Copyright:`	internationalizable copyright string
`License`	internationalizable license information. Optional
`Splash`	pre-install, post-install, pre-uninstall, post-uninstall
`Depend` NOTE: Each dependency must be on its own line. See See Package Dependency Model on page 11. for more information.	package dependencies. for example, `vendor:package.` The package won't show up in the new or updates pages if these dependencies aren't met. Here's what's expected: `vendor:package` vendor-package must exist. `vendor:package !` vendor-package must not exist. `vendor:package <=> version` vendor-package is less than, equal to, or greater than the specified version number. `vendor:package != version` vendor-package not equal to version.
`VisibleDepend` NOTE: Each dependency must be on its own line. See Package Dependency Model on page 11 for more information.	just like Depend except that the package will show up in the new or updates lists even if dependencies aren't met.

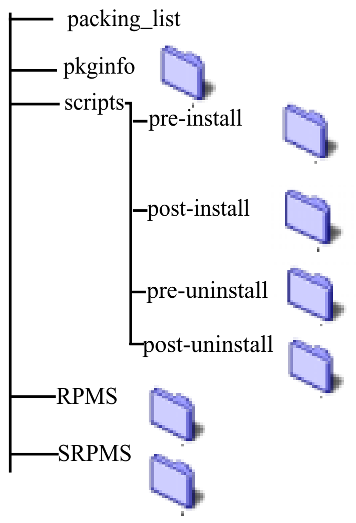
`Obsoletes`
NOTE: Each obsoletes must be on its own line. See See Package Dependency Model on page 11. for more information.
	obsoletes vendor-packages
	`format:`
	`vendor:package`
	`vendor:package <=> version`

`RPM`	used only by the actual package
`SRPM`	used only by the actual package

Note
Internationalized strings are in the following format: `[[vendor]]`. If you are specifying strings within the `pkginfo` locale directory, then do not specify a domain. Sausalito specifies the domain for you. `pkginfo` locale strings cannot include locale tags within locale tags. You can include locale tags that refer to other domains.

Package files have the following structures. Figure 1 shows the package file structure.

Figure 1 Package File Structure



See Figure 5—1, Module File Hierarchy, on page 15 for a more complete file hierarchy.

Note
The `packing_list` format for packages is very similar to the package part of the `package_list` update server packing list. You can use them interchangeably with the caveat that some fields are unused. For example, the update server information uses the `size` field. The packing list uses RPM, SRPM, and `fileName`.

The following features are only used by software update notification mechanism (**BlueLinQ**):

- `Size` (in bytes)
- InfoURL
- Location
- PackageType

The following fields are only used by actual package installation mechanism:

¥ `RPM`

¥ `SRPM`

¥ Options

9.1 Package Dependency Model

The dependency model allows you to restrict packages to particular Cobalt products, for example, the Qube 3. You can also include dependencies on other software packages. Finally, you can declare old packages obsolete.

The format for dependency requires that each dependency is on a separate line with a label denoting the type of dependency. Sausalito offers three types of dependency information:

- `Product:` Cobalt Product Dependency such that the package will install if other software products that are needed are not already installed. These are checked by product ID, for example `4000WG.` You can use a specific product, particular version, or you can use a Perl regular expression here.
- Package dependencies:

  ¥ `Depend:` Normal package dependency based on the version number being less than (<), equal to (=), or greater than (>) the version number specified.

  ¥ `VisibleDepend:` Visible dependency: same as `Depend` but is only useful for the software update mechanism. The packages that do not meet dependencies behave identically to the `Depend` in all other manners to new or update packages despite the fact that the package can t be installed.
- `Obsoletes:` Obsoletes packages name or name and optional version, less than (<), equal to (=), or greater than (>) the version number specified, which removes information about other packages of that name or version number specified.

10 Information for Installing Stand-alone Packages

The following are used in the actual package installation process but not in update server-supplied information. They are not used for the update server `pkginfo`.

- RPM
- SRPM
- `Options` (in a comma-separated list) include:

  ¥ `reboot`

  ¥ `refreshui`

  ¥ `refreshcce`

  ¥ `uninstallable`
- These fields are used to provide information and are included in the actual package as well as provided by the update servers:
- Package identification

¥ `Name and nametag`

¥ `Version and versionTag`

¥ `Vendor and vendorTag`

- Description

¥ `shortDesc`

¥ `longDesc`

- License information

¥ `License`

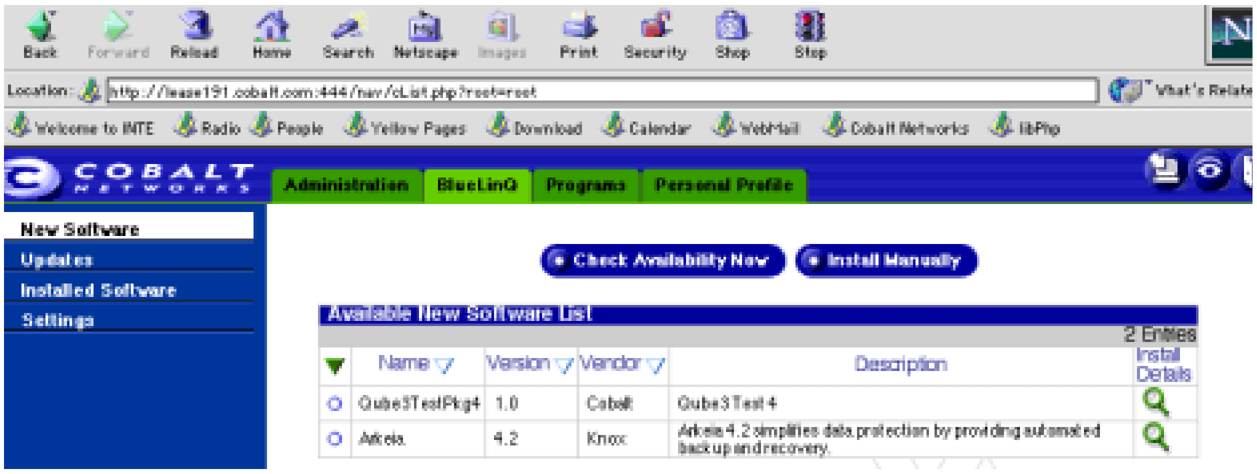
¥ `Splash`

- Category

These fields are found only in update server package:

- `Size` (in bytes)
- `PackageType`: `complete` or `update`
- Location
- `InfoURL`: a pop-up window appears when the user clicks the magnifying glass

Figure 2 New Software Installed



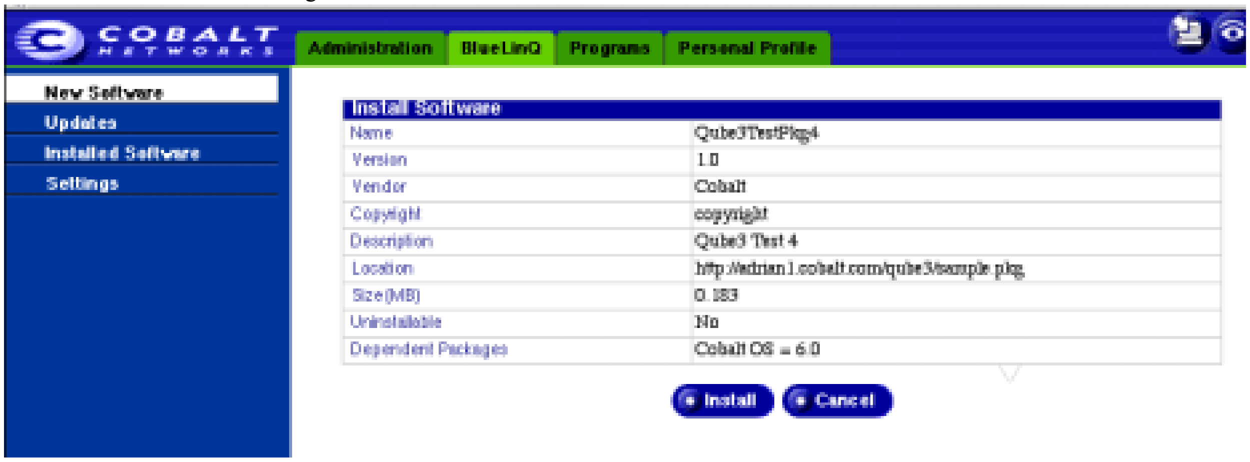
Available New Software List

2 Entries

	Name	Version	Vendor	Description	Install Details
○	Qube3TestPkg4	1.0	Cobalt	Qube3 Test 4	🔍
○	Arkeia	4.2	Knox	Arkeia 4.2 simplifies data protection by providing automated backup and recovery.	🔍

- If you click on the magnifying glass, you see the information shown in Figure 3, which corresponds to the information in Table 3, “Package List Format,” on page 8.

Figure 3 New Software Installation Details



Install Software	
Name	Qube3TestPkg4
Version	1.0
Vendor	Cobalt
Copyright	copyright
Description	Qube3 Test 4
Location	http://adrian1.cobalt.com/qube3/sample.pkg
Size(MB)	0.183
Uninstallable	No
Dependent Packages	Cobalt OS = 6.0

10.1 Software Update Server

Note
If the `infoURL` file exists, it displays a popup window and will not install the actual package.

The **BlueLinQ** tab on the Qube 3 has an **Updates** menu. This page lists available software with the following information.

- Update server-provided information (name, vendor, locale, description)
- Pop-up information. `InfoURL` displays the URL to be passed the Qube’s serial number
- The package checks for an `InfoURL`. If one exists, the page referenced by the `InfoURL` appears. If not, the package presents the license information, and installs after the user accepts the license agreement.

When users click on **Install Details**, the Qube 3:

- Displays the splash page if there is one or displays a license agreement in standardized license format.
- Begins installation

When the user begins installation, these events occur on the Qube 3:

- It checks for a signature and attempts to authenticate it, if one is present. If the signature cannot be authenticated, a message is displayed letting the user know that the signature check failed.
- It runs the preinstallation script.
- It installs the Redhat Package Modules (RPMs.

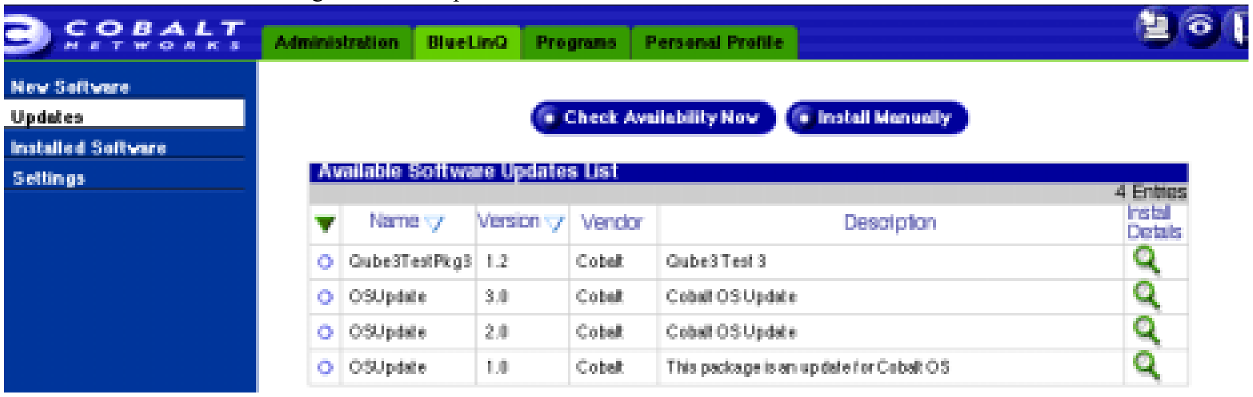
Note
Cobalt Networks uses Redhat Package Manager (RPM) files because applications are easy to manage if they are installed using RPM utilities. For details on creating *.rpm files (also

known as "redhat package module" files), see *Maximum RPM*, by Marc Ewing and Erik Troan. *Maximum RPM* is the definitive technical reference for the RPM packaging system; it provides information on RPM's history, usage, and internals from both the user and packager perspectives. Also, see `http://www.redhat.com/` for the most up-to-date information about RPM technology.

- It runs the postinstallation scripts.
- It reboots or refreshes, if those options are set.

Figure 4 shows the Update Server page.

Figure 4 Update Software Installed



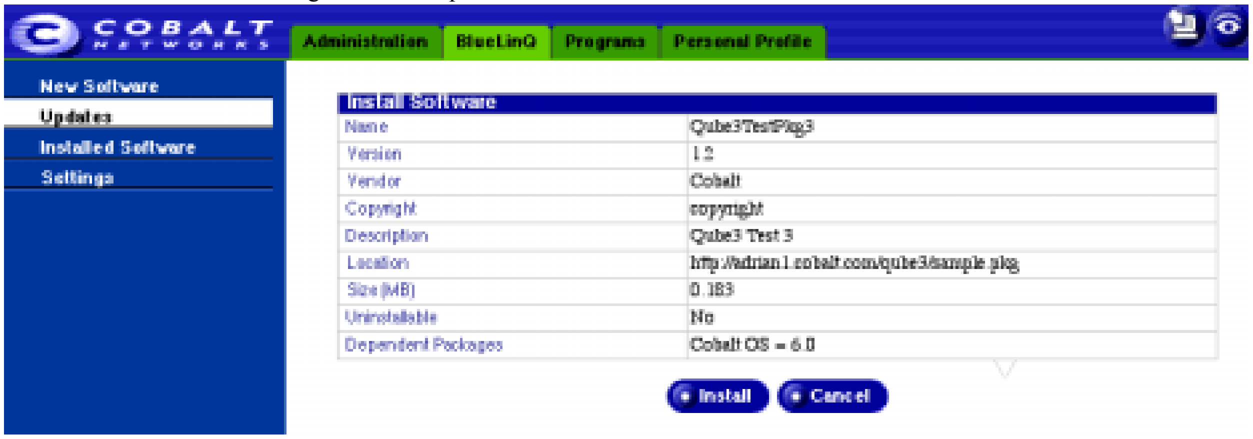
Available Software Updates List

4 Entries

	Name	Version	Vendor	Description	Install Details
○	Qube3TestPkg3	1.2	Cobalt	Qube3 Test 3	🔍
○	OSUpdate	3.0	Cobalt	Cobalt OS Update	🔍
○	OSUpdate	2.0	Cobalt	Cobalt OS Update	🔍
○	OSUpdate	1.0	Cobalt	This package is an update for Cobalt OS	🔍

If you click on the magnifying glass, you see the information shown in Figure 5, shown in Figure 3, which corresponds to the information in Table 3, Package List Format, on page 8.

Figure 5 Update Software Installation Details



Install Software

Name	Qube3TestPkg3
Version	1.2
Vendor	Cobalt
Copyright	copyright
Description	Qube3 Test 3
Location	http://adrian1.cobalt.com/qube3/sample.pkg
Size (MB)	0.183
Uninstallable	No
Dependent Packages	Cobalt OS = 6.0

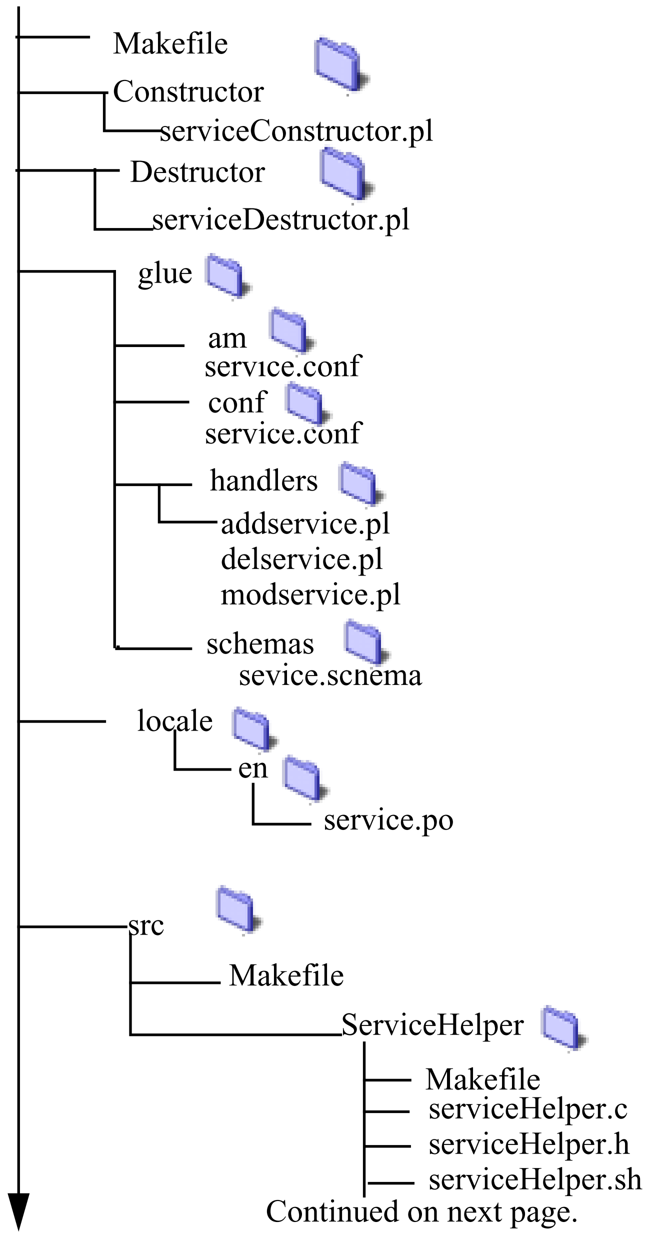
11 Development Details

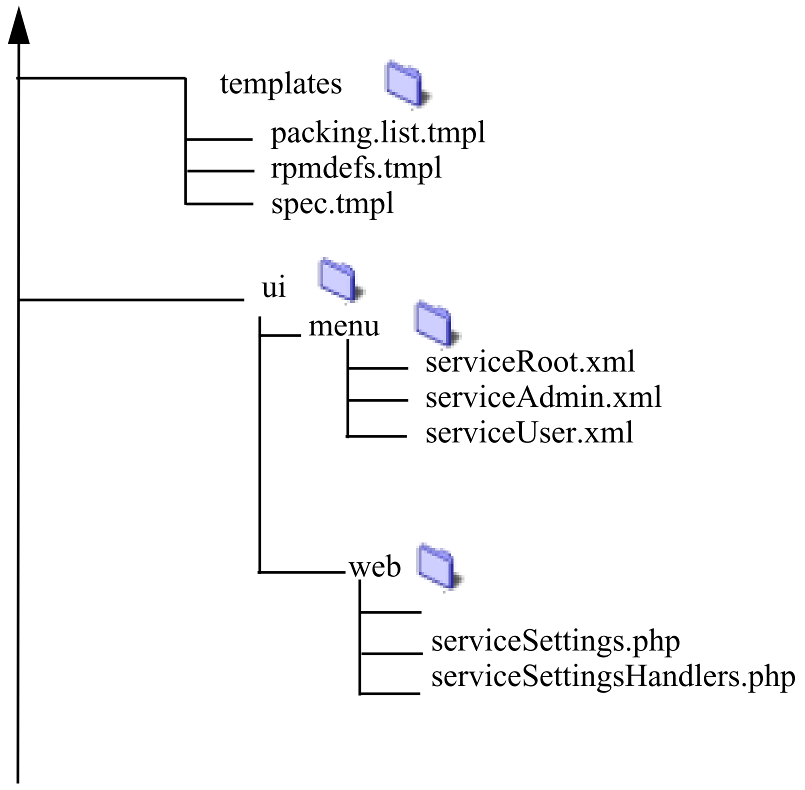
Modules expect the following auxiliary support from Sausalito development tools:

- `SAUSALITO/devel/module.mk` for all the `Makefile` rules.

- `SAUSALITO/bin/mod_rpmize` for the `RPM` specification file generator.

Figure 5—1 Module File Hierarchy




templates
packing.list.tmpl
rpmdefs.tmpl
spec.tmpl
ui
menu
serviceRoot.xml
serviceAdmin.xml
serviceUser.xml
web
serviceSettings.php
serviceSettingsHandlers.php