E·FUNDS℠

Oasis®

Packages Programmers Guide

Packages Programmers Guide

This publication was produced by eFunds Canada Corporation.
Published in Canada.

Comments are welcome and may be sent to

eFunds Canada Corporation, Publications Department
20 York Mills Road, 4th Floor, Box 700
Toronto, Ontario
M2P 2C2 Canada

Revision History

Date of Issue	Document Id	Document Version	Software Version
May 2000	PK10R720	1.0	7.2.0
Feb. 2001	PK11R720	1.1	7.2.0[a]
June 2002	PK12R731	1.2	7.3.1[b]
May 2004	PK12R731	1.2	7.3.1[c]

a. Extending SDK Releases section included.
b.Updated the section Extended SDK Releases.
c.Updated title page and copyright notice.

Table of Contents

Part III: Technical Reference

Appendix A:

List of Figures

Preface

Purpose of this Document

The packaging initiative addresses a much-needed reorganization of the product source directories into a more coherent structure based on product modules called packages.

Since this new organization has a radical impact on how the future products of eFunds Corporation will be marketed and how customers will use the new package structure to build their own products, the Packages Guide assists the user both to understand the new scheme and how to use it to best advantage. All definitions of the new packaging structure are designed to be adequate and comprehensive. All relationships within and between packages cannot be documented, but instructions on how to obtain information concerning these relationships is supplied.

How this Manual is Organized

This document is structured into three main components, or *Parts,* whose content is targeted to different audiences. These modules are:

High-level overview - provides a technical introduction to the features and benefits of the new Packaging structure and environment, and an overview of the architecture of the new format. This module is intended for business analysts and the general readership.

Application-level module – this module provides practical examples of how to use the new development environment from a programmers perspective. It is intended as a how-to document to assist application developers, system developers, business analysts, and others who need to build and maintain applications in a practical and hands-on manner.

Technical reference – this module provides any technical specifications for the packaging environment, such as lists of variables and function symbols, for example. This module is intended for a technical audience.

Related Documents

Name	Description
IST/Foundation Administration Guide	Describes the original IST/Foundation development environment and tools.
IST/Switch Administration Guide	Describes the installation procedures for the products.

Part I: Introducing Packages

Section 1: Packages and Products

1.1 Motivation

The source directories in the eFunds Corporation development environment have been reorganized from the *Generic* and *Product* trees to a package-based peer structure. The intentions behind this reorganization are:

- to rationalize the development environment and make it more efficient and economical in the use of the system resources required to develop, maintain, and build the products;
- to disentangle product components from extraneous dependencies and thereby create more robust, maintainable, and re-usable code;
- to provide a logical re-structuring of eFunds Corporation products into more market-oriented packages or groups of packages.

The packaging initiative reorganizes the product source directories into a more coherent structure based on product modules called packages. Each package consists of components, consisting of source code, binaries, libraries, documentation, tools, and other elements consistent with a necessary and sufficient item list defining the package, and from which the package can be built, installed, and tested in a consistent and reproducible way.

Dependencies between package components are contained within the package. Each package has minimal dependencies on any other package. Dependencies between packages are isolated and expressed in terms of a known interface. A developer may check-out and build a package from the source repository, modify the components, and rebuild the product of the package without needing to touch other unmodified packages. This will reduce both the time and resource usage required to maintain each local development environment.

1.2 Previous Environment

A summary of the previous development environment is described below:

"Directories are partitioned under /oasis into generic, product and site subtrees:

generic	*contains common sources, libraries and binaries which are used over a number of different products (such as mailbox functions);*
product	*contains sources, libraries and binaries for all products, namely IST/Switch, IST/Foundation, IST/Card, IST/ Interchange, and Transaction Simulator (TRSIM);*
site	*contains site specific customizations made on top of standard generic and product platforms for specific customer sites. All project development activity is restricted to the site subtree.*

Three UNIX environment variables, $OGENERIC_ROOT, $OPRODUCT_ROOT, and $OSITE_ROOT define the root directories for generic, product and site context. These must be set before using any of the script operations. This requirement is enforced prior to the execution of any script.

Development and release directories are immediately under $OGENERIC_ROOT, $OPRODUCT_ROOT and $OSITE_ROOT. Code development takes place in the "dev" subdirectory of the product in question. Minor and major release levels are contained in appropriately numbered peer directories."

1.3 New Package Environment

Under the new package architecture, there are four main types of directory in an environment: the source repository, the source directory, the build directory, and the prefix directory.

1.3.1 Source Repository

The source repository is the physical storage of all versions of tested source files of all versions of tested package implementations. Currently, the */Osrc/repository* directory is the source repository.

The source repository should not be used as a working directory. Instead, developers should work in the source directory after this is connected to the source repository with the *opc primedir* command.

1.3.2 Source Directory

The source directory contains all the source files and configuration information for all packages. It is the directory where compilers locate the source code.

New source files are created in a source directory and are snapshot into the source repository after they have been tested.

Existing source files are staged from the source repository onto a source directory.

Movement of source files between the source repository and source directory is handled by the *opc* (Oasis Package Control) command.

1.3.3 Build Directory

The build directory contains the object, library and executable files. These are intermediate files that are created during compilation.

1.3.4 Prefix Directory

The prefix directory contains the end result of a build. It is the location where selected source files (interface header and sql files) and product files (library and executable files) are copied during the build.

The prefix directory layout is a flattened runtime environment. Its intent is to reduce the effort to test and install the software. It becomes the major part of a binary release sent to customers by Configuration Management.

A prefix directory can become a baseline if it is built from a known release.

While the build and prefix directories are specific to a platform, the source directory is platform-independent. It follows that a developer should create separate build and prefix directories for each supported platform.

The package implementations are staged under the source directory. The structure of the source directory is illustrated in Figure 1 on page 3.

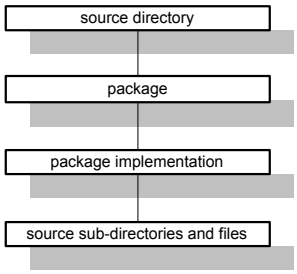


Figure 1: Source Directory Structure

At the highest level there is a single source directory.

Beneath the source directory are the sub-directories for each package. For example, there is one sub-directory for 'mbox_util' and another for 'port_util'.

Beneath each package directory is the sub-directory for each package implementation. For example, beneath the package directory 'mbox_util', there is the sub-directories 'mbox_util-0.1.9'.

Beneath each package implementation directory are the source files that implement the package. The source files may be located in the directory or in one of its sub-directories.

The diagram below depicts a source directory of package implementations that are required to build the executable 'mb'.

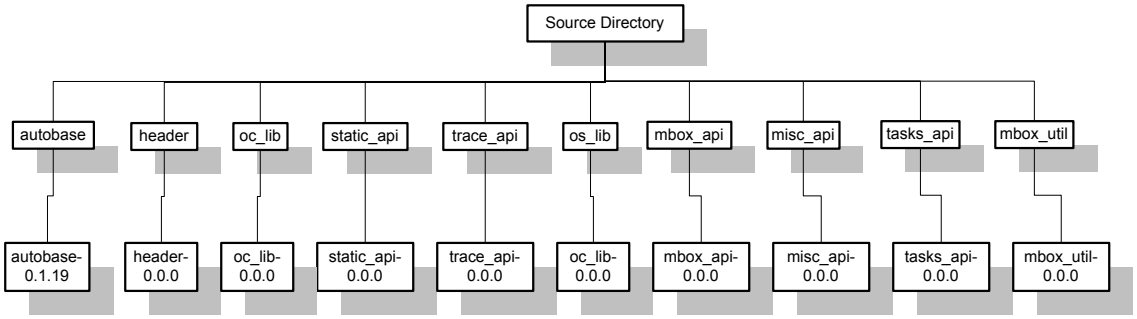


Figure 2: Source Directory Example

Section 2: Package Definition

2.1 Package Characteristics and Implementation

A package is a unit that provides some identifiable functionality. Each package has a name, and each name must conform to these naming conventions:

- a package name cannot be one of the reserved names: *autobase*, *header*, *include*, *bin* and *lib*. Nor can it be one of *forms, sql*, *cfg*, *istdir, ositeroot,* or *tmp*.
- the name must not be in use by another package;
- the name must contain a suffix element such as *_lib*, *_api*, *_util,* or *_driver*. The first two are reserved for library packages; that is, packages that produce no executables, except those for testing purposes. The choice of *_lib* or *_api* is left to the developer.

For example, *autobase*, *mbox_util* and *dbm_api* are names of different packages.

A package is not a physical entity. What does exist physically is the package implementation - the collection of files that implement the functionality defined by the package.

Each package implementation is assigned a version number. The name and version number together uniquely identify a specific implementation of a package. The version numbering scheme adopts the following convention:

<major-interface#>.<minor-interface#>.<patch#>

The interface of a package is its public header file. This interface includes the collection of the public variables, classes, types and prototypes of functions defined by the package. An interface is identified by the major-interface number and the minor-interface number pair.

A major interface is a subset of an interface that all package implementations with that major-interface number must define.

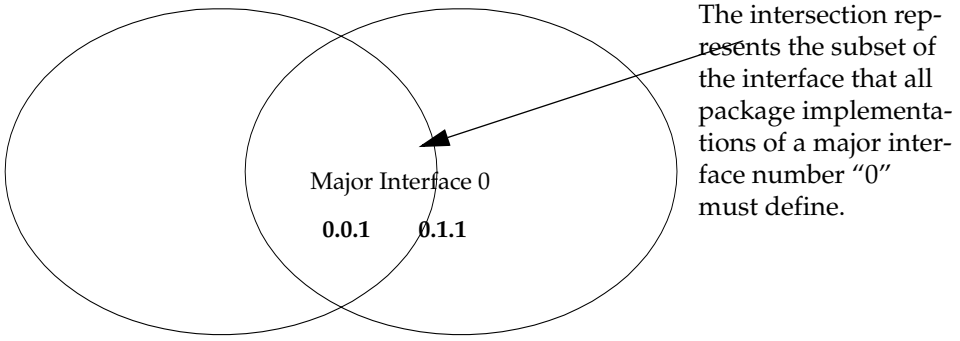


Figure 3: Major Interface

A minor interface number identifies the extension a package implementation may have to the major interface.

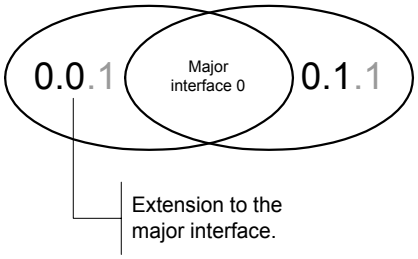


Figure 4: Minor Interface

Example

The major and minor interface numbering can be clarified with the following example, which follows the evolution of the interface of a package. Note that 'X' is used for the patch number since we are concerned in this example only with the package interface.

1. Package implementation 0.0.X. Create the first implementation of the package.
2. Package implementation 0.1.X. Add functions to the interface on top of 0.0.X, as the package evolves.
3. Package implementation 0.2.X. Add functions to the interface on top of 0.1.X, as the package evolves.
4. Package implementation 0.3.X. Add functions to the interface on top of 0.0.X, as a result of an emergency patch release.
5. Package implementation 0.4.X. Merges the interface extension in 0.2.0 (a superset of 0.1.0) and 0.3.0.
6. Package implementation 1.0.X. Removed some obsolete functions.

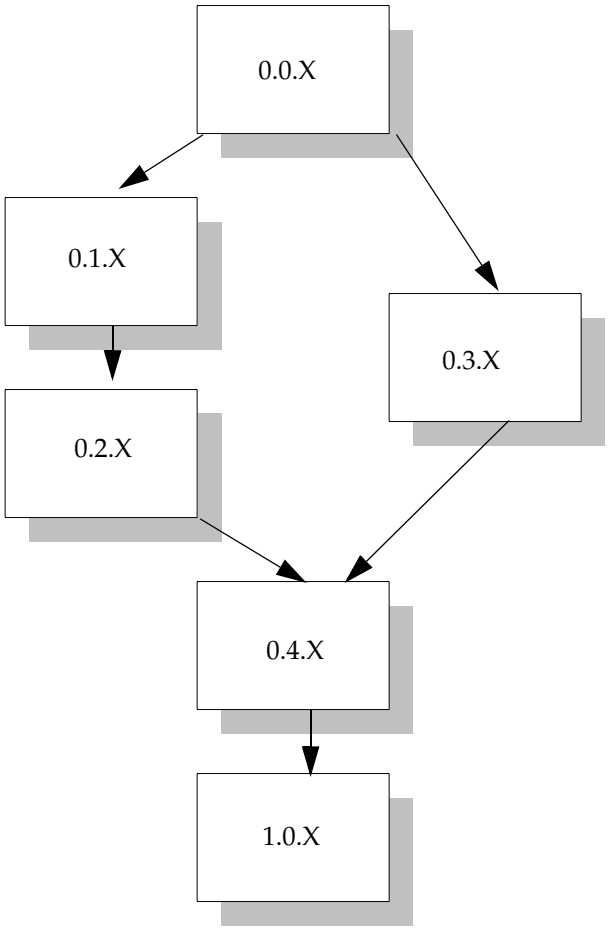


Figure 5: Evolution of a Package Interface

The package implementation in the example is created with interface number 0.0 (i.e. major# 0 and minor# 0). A change request incurs extra functions.

A patch number identifies a particular implementation of an interface (major and minor) of a package. Different implementations of a particular interface should only differ by their patch number.

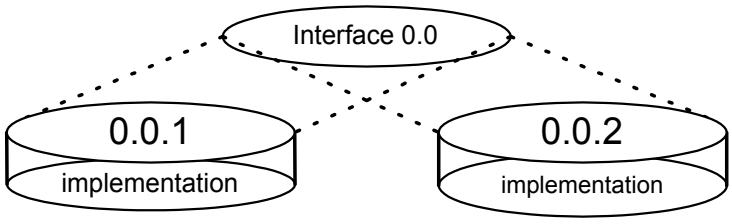


Figure 6: Interface Patches

For example, *autobase-0.1.19*, *mbox_util-0.0.0* and *dbm_api-3.0.0* are different implementations of different packages. In particular, *mbox_util-0.0.0* is the first implementation of interface 0.0 (i.e. major# 0 and minor# 0).

Each package produces a number of product files. These can be header, library or executable files. Library files are assigned version numbers.

Both packages and their contents share the same name space. Although the suffix rule that has been imposed on packages makes collision between package and element names very unlikely, care must be observed to prevent name collision.

No package should produce executables with names like “mbox_util” or “shc_util” because these are package names.

Version numbers have the same meaning to the library files as they do to the packages, only at a smaller scale. Consider the interface of a package to be made up of the interface of individual library files, and the version number of a package is the result of rolling-up the version number of individual library files.

For packages that produce one library file, the version number of the library file is identical to that of the package. In the packages that produce more than one library file, the version number of the package will change according to one of the following rules:

1. Increment the package major number and start the minor number and patch number from 0, if the major number of any library file has been changed.

2. Increment the package minor number and start the patch number from 0 if the major number of the package has not been changed and the minor number of any library file has been changed.

3. Increment the package patch number if neither major number nor minor number of the library has changed.

Each package has dependencies on a number of elements. These elements may be the header, library or executable files produced by the package itself, or by other packages.

When defining dependent libraries in the makefile.am file (to satisfy the linkage of a library or an executable), you can specify the following:

- a specific implementation of the library: for example, *lotrace-0.0.0* for *libotrace-0.0.0*;
- a specific interface of the library: for example, *lotrace-0.0* for any *libotrace-0.0.x*;
- a specific major interface of the library: for example, *lotrace-0* for any *libotrace-0.X.Y.*

Note: for a library *liboA.X.Y.Z*, this will be pointed to by three other links – *liboA*, *liboA-X*, *liboA-X.Y.* When a library is required at the interface on a major interface level, it is accessed through these links.

For example, package implementation *mbox_util-0.0.0* depends on: header, *libocatsig-0, libomisc-0, liboipc-0, libotrace-0, libocfg-0, libosyslg-0, libocmd-0, libotext-0, libotasks-0, libombox-0, libostring-0 and libobrand-0*. The '-0' suffix of the libraries is the major-interface number required by the package *mbox_util-0.0.0*.

Part II: Using Packages

Section 1: Producing Binary Files

1.1 Introduction

Once the source files comprising the package implementations are stored under the source directory, they must go through a number of processes before libraries and executables can be generated. The steps involved are illustrated in the following diagram.

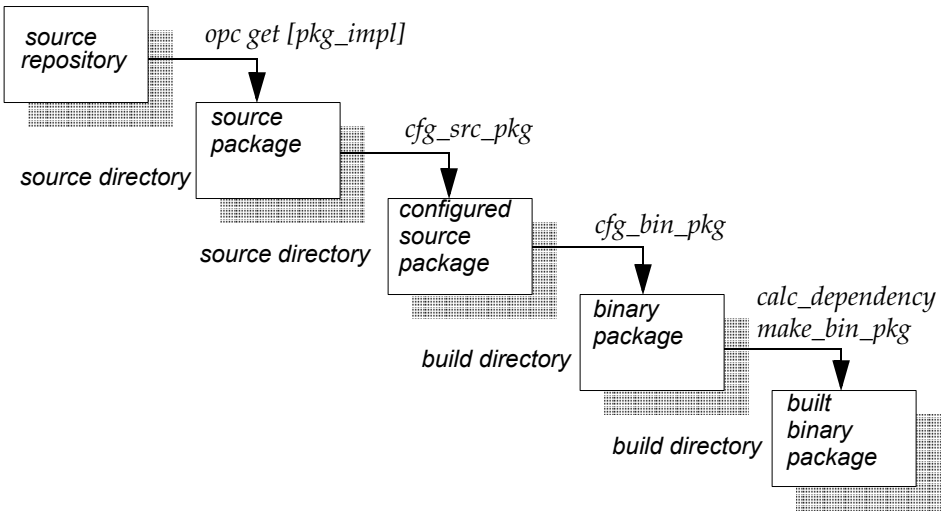


Figure 7: Package Processing

1.2 Configuring Source Packages

The next activity is to configure the source packages – the source files of a package implementation under the source directory. The source files obtained from the source repository may come without makefiles. You must first create the configuration scripts which are required to generate the Makefiles.

This step is performed with the *cfg_src_pkg* program and requires a package implementation of *autobase*.

1.3 Configure Binary Packages

The next activity is to run the configuration scripts in the build directory to generate the platform specific Makefiles. In short, the configuration scripts run a series of tests to determine, on the current platform, the compiler or linker command or option to be used, and the presence or absence of specific header-files, libraries, and functions.

The command option *--dependent* can be specified to include both the specified package and also those that the package depends on. For example:

```
cfg_bin_pkg --dependent mb
```

executes the configuration scripts of all packages required to build the executable 'mb'. The result is the creation of a series of binary packages (i.e. the files created for a package implementation under the build directory) in the build directory and the *Makefile*s in each package. The following diagram depicts a build directory of directories (or binary packages) that result from this command.

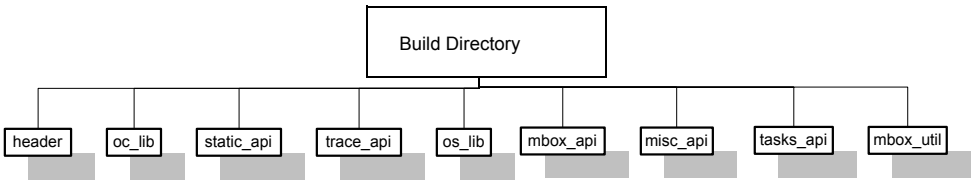


Figure 8: An Example of a Build Directory

There is a one to one mapping between each package directory under the source directory and that of the build directory - with the exception of *autobase*, which only exists in the source directory.

Note: *cfg_bin_pkg* needs to know where the source directory is. It can be told explicitly where it is with the option *-srcdir=<some-directory>*. Otherwise, the path is obtained from the environment variable DEFAULT_SRCDIR, if defined. If not, it is assumed to be '/Osrc/src'.

1.4 Builld Binary Packages Using Make

The next activity is to run the *Makefiles* under the binary directory and produce object, library and executable files. This step is performed with the utility *calc_dependency* running from the binary directory. The result is the creation of three special directories under the binary directory: *include*, *lib* and *bin*. These

directories house respectively the public header files, library files and executable files either copied or produced from the packages. For example, the command

```
calc_dependency --build=makeinstall mb
```

executes the *Makefiles* of the binary packages that produces executable 'mb' in the reverse order of the dependency (i.e. the packages are "made" before those that depend on them). The script *calc_dependency* will attempt to build the packages in parallel if possible.

The diagram below shows the addition of the *include*, *bin*, and *lib* directories after the binary packages are made.

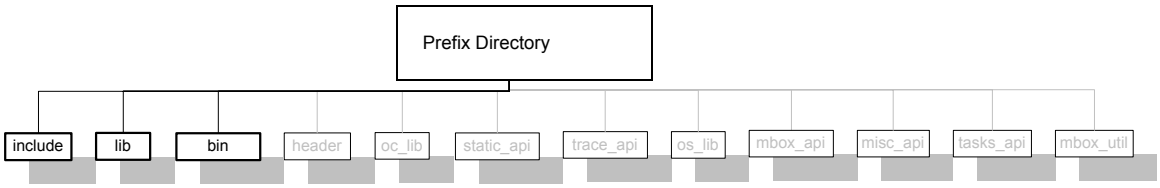


Figure 9: Example of Prefix Directory Following Make Execution

An alternative to *calc_dependency* to make the binary packages is to use the shell script *make_bin_pkg*. This command allows you to specify exactly which binary packages are to be made. For example, the command

```
make_bin_pkg os_lib trace_api
```

makes only the *os_lib* and *trace_api* packages.

Another situation where you might want to use *make_bin_pkg* instead of *calc_dependency* is when the build must be executed in serial fashion. You may want to do this if you want to analyze the output. For example, the command

```
make_bin_pkg -dependent mb
```

executes the *Makefiles* of the binary packages which produce the executable **mb** one package at a time.

Section 2: Packages – Examples of Use

2.1 Introduction

This section presents practical examples to illustrate the use of the packages build environment, source repository, and tools. Some commands are aliased during the environment generation. See Command Aliases on page 46.

Online help is available through the following command:

```
. /Osrc/tools/bin/pkghelp
```

2.2 Example 1: Build liboc Library

In this example you will build the liboc library which contains the OcTypes member classes. The steps are listed below. First, you need to setup your working environment.

1. Setup your development environment:

   1.1 You need to set your $PATH variable to include the GNU and internal tools directories. You can perform this manually, or by running the following command:

   ```
   . /Osrc/tools/bin/pkgstart
   ```

   1.2 You need to define the installation directories for your vendor database to the appropriate environment variable. For example:

   ```
   export ORACLE_HOME=/apps/oracle/product/8.0.6
   ```
   or
   ```
   export INFORMIXDIR=/apps/informix/se/7.24
   ```

   1.3 Select your working directories. You should setup private directories to avoid conflicts with the work of other developers. Setup a directory under `/Osrc`. This will contain one source directory and multiple build and prefix directories. The example chosen in this section is `/Osrc/example`.

   The directory structure in Table 1 on page 14 is an example of the type of structure you should create.

Table 1: Example Packages Directory Structure

Directory	Path
Source repository	`/Osrc/srcrepository`
Source directory	`/Osrc/example/srcdir`

Table 1: Example Packages Directory Structure (Continued)

Directory	Path
Build directory	`/Osrc/example/bdirSOL`
Prefix directory	`/Osrc/example/pdirSOL` [a]

a. pdirSOL and bdirSOL refer to directories specific to the build environment for the Solaris platform. Directories for other build environments, such as HP-UX, would have a different suffix – in the case of HP-UX, this would be pdirHP and bdirHP.

1.4 Set your environment

Execute the following commands to create the directories.

```
mkdir -p /Osrc/example/srcdir /Osrc/example/bdirSOL /Osrc/example/
pdirSOL
```

Once the working directories have been created, they need to be declared. Execute the following command to set the $PATH variable to include them:

```
eval 'gen_env /Osrc/example/srcdir /Osrc/example/pdirSOL /Osrc/
example/bdirSOL /Osrc/baseline/f.sun27.t3.oracle.d'[1]
```

Note: It is important to specify the prefix directory before the build directory.

1.5 Verify your working environment. The program *chk_pkg_env* verifies your current environment and generates warnings and errors if the environment is incorrect. It checks the following settings in your environment:

- the presence of the required compiler/linker;
- that the shell interpreters exist in the expected location;
- that the Perl interpreter is correctly located.
- verifies if the macro preprocessor M4 exists and comes from GNU;
- checks if the GNU tools *autoheader*, *automake*, and *autoconf* can be located;
- checks the validity of the environment variables for the working directories;
- checks the validity of the database environment variables.

1.6 Execute the following commands to change the directory to the source directory, and to connect the source directory to the source repository:

```
repo
opc primedir /Osrc/example/srcdir
```

1. If baseline is used (eg. /Osrc/baseline/f.sun27.t3.oracle.d - the 3rd generation foundation release with oracle and debug enabled)

This step creates soft-links in the source directory which point to the source repository. The SCCS files of the source files are accessed, and can allow source control operations to be performed through these links.

A special package called *autobase* is always required. You need to execute the *opc* command to determine the latest version of *autobase*. Unless you are trying to reproduce an old environment, always use the latest implementation of *autobase*.
Execute the following commands to get the latest implementation of *autobase:*

```
sdir
opc get `opc latest autobase`
```

This returns the following result in this example:

```
1.1 autobase-0.1.15 [new package]
```

2. Select your packages

   Execute the following commands to determine which packages are required to build the liboc library. The *calc_dependency* command queries and updates the package database. See calc_dependency on page 47.

   ```
   calc_dependency liboc
   ```

   The *calc_dependency* command will return information about the package versions and any dependencies between them. For example:

   ```
   header-0.0.0
   oc_lib-0.1.0
   ```

3. Stage your packages

   Execute the following commands to stage the selected packages to the selected source directory:

   ```
   sdir
   opc get header-0.0.0 oc_lib-0.1.0
   ```

   See Figure 13 on page 71. Also Figure 14 on page 72.

4. Check the source directory

   Run the following commands to check the state of the packages:

   ```
   sdir
   opc info autobase header oc_lib
   ```

   The output looks like this:

   ```
   Information of package autobase

   1.1        autobase-0.1.15      [new pkg]

   1.2        autobase-0.1.16      [Added new script pkg_name.]
   ```

```
1.3       autobase-0.1.17     [Modified so that developer can
                              assign version to pkg before it
                              is snapshot.]

1.4       autobase-0.1.18     [Added validation of duplicated
                              product from different packages.]

1.5       autobase-0.1.19     [Ensure new subdir is added to
                              SCCS-package when delgetting
                              package.]

1.6       autobase-0.1.20     [Added support for oracle ProC +
                              Added script expsql to calc
                              dbparse output files.]

1.7       autobase-0.1.21     [Added release management script
                              pkgrel]

1.8       autobase-0.1.22     [ensure the Makefile under diff
                              subdir is created in the right
                              order]

1.9       autobase-0.1.23     [Added opc cmd editfile & unedit-
                              file to facilitate the modifica-
                              tion of the current version of
                              file instead of that of the lat-
                              est.]

1.10      autobase-0.1.24     [Added baseline support.]

1.11      autobase-0.1.25     [Fixed bug in baseline support]

1.12      autobase-0.1.26     [Sorted pkg info by numeric
                              order.]

1.13      autobase-0.1.27     [Added support for tar pkg bina-
                              ries + Added alias in gen_env +
                              Improved sort algo of pkg impl-
                              mentations.]

1.14      autobase-0.1.28     [XXXXXXX; Added chk-sum & size in
                              listing + Added pkg-list filter-
                              ing tool]

1.15      autobase-0.1.29     get  [XXXXXX; Added search path
                              for cpp]

No implementation has not been checked out.

---

--- Information of package [header] ---

1.1       header-0.0.0        get   [new package]

No implementation has not been checked out.

--- Information of package [oc_lib] ---

1.1       oc_lib-0.1.0        get   [new pkg]

No implementation has not been checked out.
```

Also, examine the structure of the source directories under `/Osrc/example/srcdir`.

5. Configure your source packages

   Run the following commands to produce the work-files that are used to generate the Makefiles:

```
sdir
csp header-0.0.0 oc_lib-0.1.0)
```

   See Configure Source and Binary Packages on page 73 for a graphical representation of the environment built by these commands.

   The output looks like this:

```
header === Configuring source package [ header ]
header === Using autobase directory [ autobase/autobase-0.1.29 ]
header === Configuring package [ header ] version [ 0.0.0 ]
header === Installing configure.in ...
header === Installing aclocal.m4 ...
header === Copying libtool/install script ...
header === Running autoheader ...
header === Running automake ...
header === Running autoconf ...
header === Generating configure_fast ...
header === This source-package is now updated with the latest
header     version of configuration scripts. Now, please run
header     'cfg_bin_pkg' and 'make_bin_pkg' on the affected binary
header     packages.
oc_lib === Configuring source package [ oc_lib ]
oc_lib === Using autobase directory [ autobase/autobase-0.1.29 ]
oc_lib === Configuring package [ oc_lib ] version [ 0.1.0 ]
oc_lib === Installing configure.in ...
oc_lib === Installing aclocal.m4 ...
oc_lib === Copying libtool/install script ...
oc_lib === Running autoheader ...
oc_lib === Running automake ...
oc_lib === Running autoconf ...
oc_lib === Generating configure_fast ...
oc_lib === This source-package is now updated with the latest
oc_lib     version of configuration scripts. Now, please run
oc_lib     'cfg_bin_pkg' and 'make_bin_pkg' on the affected binary
oc_lib     packages.
=== All 2 Jobs Ended
```

6. Configure binary packages

   Run the following commands to produce the Makefiles:

```
bdir
cbp header-0.0.0 oc_lib-0.1.0)
```

   A graphical representation of the environment built by this command is given in the Appendix. See Figure 15 on page 73.

   The output from this command is extensive and is not included here.

7. Make binary packages

   Run the following commands to perform a *make install* with the Makefiles:

```
bdir
mbp header-0.0.0 oc_lib-0.1.0
```

2.3 Example 2: Modify Package oc_lib-0.1.0

This section illustrates how to modify an existing package using the example of the oc_lib package. You need to set up your directory structure, set your environment variables, and select packages as explained earlier. See Example 1: Build liboc Library on page 14, steps 1 to 3.

1. Check-out package oc_lib.

```
sdir
opc edit oc_lib-0.1.0
```

2. Check source-dir

   Execute these commands to check the state of package oc_lib-0.1.0:

```
sdir
opc info autobase header oc_lib
```

3. Modify file oc_time.cpp

```
sdir ; x oc_lib-0.1.0
opc editfile oc_time.cpp
vi oc_time.cpp
opc delgetfile oc_time.cpp
```

4. Configure source packages

   Execute the following commands to renew the work-file that is used to generate Makefile's in package oc_lib-0.1.0:

```
sdir
csp oc_lib-0.1.0
```

   Note that the *-dep* option is not needed since only one package is affected.

5. Configure binary packages

   Execute these commands to produce Makefiles:

```
pdir
cbp oc_lib-0.1.0
```

6. Make binary packages

   Execute these commands to perform a *make install* with the Makefiles:

```
pdir
mbp oc_lib-0.1.0
```

7. Rename (add + delete) file oc_decimal.cpp to oc_new.cpp

```
sdir ; x oc_lib-0.1.0
opc editfile oc_decimal.cpp
mv oc_decimal.cpp oc_new.cpp
```

```
opc unedit oc_decimal.cpp
rm -f oc_decimal.cpp
```

8. Update Makefile.am

```
cd /Osrc/example/srcdir/oc_lib/oc_lib-0.1.0/libsrc
opc editfile Makefile.am
vi Makefile.am
< do some work >
opc delgetfile Makefile.am
```

9. Compare files in the source-dir

   Execute the following commands to display which file has been changed in package oc_lib-0.1.0:

```
sdir
opc diffs oc_lib-0.1.0
```

   Note that all files must be registered for *opc diffs* to work.

10. Register package oc_lib, using opc

```
sdir
opc examine oc_lib-0.1.0
opc delget oc_lib-0.1.0
```

11. Check source-dir

    Run these commands to check the state of package oc_lib:

```
sdir
opc info oc_lib
```

2.4 Example 3: Creating a New Package

The purpose of this example is to illustrate how to create a new package. If we take as a starting point a main product of the package called library *xyz*, then the name of the package is *xyz_lib*. The steps enumerated below explain how to proceed with creation of this package.

1. Create package *xyz_lib-0.0.0*

   Run these commands to create the new package *xyz_lib-0.0.0*:

   ```
   sdir
   csp --create=yes xyz_lib
   ```

2. Distribute source files to the appropriate subdirectories

   The following table lists the files that this package should contain:

File Type	Filename	Subdirectory
Public interface header file	xyz_interface.h	include
Locally shared header file	a.h	include
Private header file	xyz.h	libsrc
Implementation files	a.cxx, xyz.cxx	libsrc
Demo / test executables	xyztl.cxx	demo

3. Create Makefile.am files

   These files must be created in the appropriate directories beneath the subdirectories of package *xyz-0.0.0*. The definitions contained therein are read by the commands listed in step 4 below and used to build the package. Use a text editor to create these files and insert the text specified under **File Contents** in the tables below.

Subdirectory	File Contents
`/Osrc/example/srcdir/xyz_lib/xyz_lib-0.0.0/Makefile.am`	SUBDIRS = include libsrc demo test

Subdirectory	File Contents
`/Osrc/example/srcdir/xyz_lib/xyz_lib-0.0.0/include/Makefile.am`	include_HEADERS = xyz_interface.h noinst_HEADERS = a.h

Subdirectory	File Contents
/Osrc/example/srcdir/xyz_lib/xyz_lib-0.0.0/lib-src/Makefile.am	noinst_HEADERS = xyz.h lib_LTLIBRARIES = liboxyz.la liboxyz_la_LIBADD = -loc-0 $(CPP_LIBS) liboxyz_la_SOURCES = a.cxx xyz.cxx liboxyz_la_LDFLAGS = -release 0.0.0

Subdirectory	File Contents
/Osrc/example/srcdir/xyz_lib/xyz_lib-0.0.0/demo/Makefile.am	bin_PROGRAMS = xyzt1 xyzt1_SOURCES = xyzt1.cxx xyzt1_LDADD = -loxyz-0

Subdirectory	File Contents
/Osrc/example/srcdir/xyz_lib/xyz_lib-0.0.0/test/Makefile.am	TESTS = xyzt1.test

Subdirectory	File Contents
/Osrc/example/srcdir/xyz_lib/xyz_lib-0.0.0/demo/xyzt1.test	../demo/xyzt1 >/dev/null 2>&1

4. Build the package

   Run the following commands to build the binaries:

```
sdir
csp xyz_lib-0.0.0
bdir
cbp xyz_lib-0.0.0
mbp xyz_lib-0.0.0
```

5. Snapshot of package *xyz_lib*

```
sdir
opc examine xyz_lib-0.1.0
opc create xyz_lib-0.1.0
```

6. Check source-dir

Run the following commands to check the state of package oc_lib:

```
sdir
opc info xyz_lib
```

Section 3: Extending SDK Releases

3.1 SDK vs. Binary Releases

eFunds Corporation products can be delivered as run-time or SDK (Software Developers Kit) releases. Both releases provide similar content and are installed in the same manner, but the availability of the *include* directory dictates the release type. A binary release does not provide an *include* directory, while an SDK release provides an *include* directory containing all header files needed for building new functionality based on standard product libraries, allowing you to extend the product by writing your own code.

In other words, the release type determines its applicability. A binary release is an end-user environment that can be customized only via configuration, while an SDK release is a development environment that can be customized via additional code development.

To obtain a more robust and efficient development environment, all eFunds Corporation products are organized in a package-based structure. When you develop your own code, it is recommended to follow the same package-base strategy.

3.2 Compiling SDK Releases

The following procedure describes the steps you must follow to compile additional code written to extend a product (See Compilation procedure on page 30).

1. Install the SDK release from the installation media. This copies over all necessary files to the installation directory.

2. To install all GNU software required to compile the source files, locate the MAKE_RUN_BETA.tar file and extract the files in a directory called /Osrc.

   If you cannot use /Osrc, extract the files in any directory and create a symbolic link to the directory, so that /Osrc/tools/bin, /Osrc/local/bin/$A, and /Osrc/local/bin/$A/bin are visible.

   To create a symbolic link use the *ln -s <source_file> <target>*. If the directory used is /abc, to create the symbolic link issue the following command:

   ```
   ln -s /Osrc /abc
   ```

Note: The baseline directory and the pdir, bdir, and srcdir directories should never have a parent-child relationship; otherwise, step 5.1 will generate a recursive copy.

---

3. Set environment variables.

   3.1 Set the $A environment variable to the UNIX platform for which you are compiling. Note that the platform must be supported by eFunds Corporation. Currently, the supported platforms are:

   ```
   _AIX4.3_risc6000
   _HP_UXB.11.00_hp
   _SOLARIS2.7_sun4d
   _OSF4.0_alpha
   ```

   3.2 Include in the $PATH environment variable the GNU tools directories (/Osrc/local/bin/$A, /Osrc/local/bin/$A/bin, /Osrc/tools/bin) and the location of the C++ compiler. These entries should be the first four entries in the path, in the listed order.

   The following path, for example, avoids errors while running chk_pkg_env.

   ```
   PATH=/Osrc/tools/bin:/Osrc/local/bin/_AIX4.3_risc6000/bin:
   /Osrc/local/bin/_AIX4.3_risc6000/lib/perl5:$PATH
   export PATH
   PATH=/usr/vacpp/bin:$PATH
   export PATH
   ```

   3.3 Set the environment variable for the installation directory of your database. For Informix, set the $INFORMIXDIR variable. For Oracle, set the $ORACLE_HOME variable.

4. Prepare the working directory structure required by the package-based architecture.

   4.1 Create a new directory for compilation purposes (for example, /efunds/BUILDTEST) and create three sub-directories: a source directory, a build directory, and a prefix directory.

   The source directory contains the source file developed to extend the product. The build directory contains the intermediate files created during compilation: object, library, and executable files. The prefix directory contains the end-result of a build: header, sql, library, and executable files.

   While the build and prefix directories are specific to a platform, the source directory is platform-independent. Consequently, you should create separate build and prefix directories for each supported platform. For example, for an AIX platform you could have:

   ```
   srcdir
   bdirAIX
   pdirAIX
   ```

   4.2 To install an implementation of the latest autobase package, locate the autobase_<version#>.tar file and extract the files under the source directory (*srcdir*). The autobase package contains a packaging tool set used to configure other packages. Install the autobase package with the following command:

   ```
   ksh install_autobase --prefix=/Osrc/tools
   ```

   4.3 Place the source you want to recompile under the *srcdir*.

5. Prepare the compile environment.

   5.1 Execute an *eval* command to declare your working directories and establish the compile environment. For example:

   ```
   eval `gen_env srcdir pdirAIX bdirAIX /dev`
   ```

   The fourth parameter (/dev) is the absolute path of the directory where your existing release of the product is installed (also called the baseline). This parameter enforces entries like "-I /dev/include" and "-L /dev/lib" to be added to the Makefile created later in the procedure, so that the appropriate files and shared objects can be found when compiling and linking.

   5.2 Verify your compile environment using the *chk_pkg_env* command. This program verifies your current environment and generates warnings and errors if the environment is incorrect. For an execution without errors, the output should look like (for an AIX platform):

```
Found HOST SunOS solardev2 5.8 Generic_108528-05 sun4u sparc
SUNW,Ultra-250 :
Found cc (/opt/SUNWspro/bin/cc). [cc: Sun WorkShop 6 2000/04/07 C
5.1]
Found CC (/opt/SUNWspro/bin/CC). [CC: Sun WorkShop 6 2000/04/07 C++
5.1]
Found cpp (/Osrc/local/bin/_SOLARIS2.7_sun4d/bin/cpp). [GNU CPP
version 2.8.1 (sparc)]
Found java (/usr/j2se/bin/java). [java version "1.3.0"]
Found javac (/usr/j2se/bin/javac).
Found /bin/sh
Found /bin/ksh.
Found perl (/Osrc/local/bin/_SOLARIS2.7_sun4d/bin/perl). [perl ver-
sion 5.00404]
Found GNU m4. [GNU m4 1.4]
Found GNU autoheader. [Autoconf version 2.13]
Found GNU automake. [automake (GNU automake) 1.4]
Found GNU autoconf. [Autoconf version 2.13]
Found GNU make. [GNU Make version 3.74]
Found $CLASSPATH contains /efunds/JavaCC/JavaCC.zip
Found $PATH (or equiv.) contains .:/Osrc/rchoi/src2/bin:/Osrc/
rchoi/pdirSOL2/bin:/Osrc/baseline/s.solardev2.latest/bin:/Osrc/
baseline/f.solardev2.latest/bin:/apps/oracle/product/8.1.6/bin:/
home2/rchoi/bin:/home2/rchoi/bin/_SOLARIS2.7_sun4d:/Osrc/tools/
bin:/Osrc/local/bin/_SOLARIS2.7_sun4d/bin:/Osrc/local/bin/
_SOLARIS2.7_sun4d:/Osrc/local/bin::/usr/j2se/bin:/usr/local/bin:/
usr/bin:/usr/lib:/opt/SUNWspro/bin:/usr/ccs/bin:/usr/sbin:/usr/
openwin/bin:/usr/proc/bin:/usr/bin/X11:/Osrc/tools/SN452-Solaris/
bin:/usr/local/Rational2002.05.00/releases/quan-
tify.sol.2002.05.00:/usr/local/Rational2002.05.00/releases/
purify.sol.2002.05.00:/istdir/bin/shell
Found $LD_LIBRARY_PATH (or equiv.) contains .:/Osrc/rchoi/pdirSOL2/
lib:/Osrc/baseline/s.solardev2.latest/lib:/Osrc/baseline/
f.solardev2.latest/lib:/apps/oracle/product/8.1.6/lib:/home2/
rchoi/lib/_SOLARIS2.7_sun4d:/usr/j2se/lib:/usr/j2se/lib/sparc/
green_threads
Found $DEFAULT_SRCDIR contains /Osrc/rchoi/src2
Found $DEFAULT_PREFIX contains /Osrc/rchoi/pdirSOL2.
Found $DEFAULT_PREFIX/bin in executable search path.
Found $DEFAULT_PREFIX/lib in dll search path.
```

```
Found $DEFAULT_BASELINEDIR contains /Osrc/baseline/s.solardev2.lat-
est:/Osrc/baseline/f.solardev2.latest
Found $ORACLE_HOME contains /apps/oracle/product/8.1.6.
Found /apps/oracle/product/8.1.6/lib in dll search path.
ERROR: $INFORMIXDIR contains invalid direcory /apps/informix/7.31.
ERROR: $SYBASE contains invalid direcory /apps/sybase/11.9.2.
```

6. Copy selected files from the package to the package you want to compile. For this purpose, execute the following command from the implementation directory of the package you want to compile, that is two levels down in the package directory.

   ```
   cfg_src_pkg
   ```

7. Issue the following commands to create the Makefiles for the package you want to build. The Makefile is created in the build directory.

   ```
   bdir
   cbpdib <package_name>
   ```

   or

   ```
   cbpdob <package_name>
   ```

   where

bdir	changes directory to the build directory.
cbpdib	is an alias executing the command cfg_bin_pkg with the debug turned on, with Informix, and the set baseline.
cbpdob	is an alias executing the command cfg_bin_pkg with the debug turned on, with Oracle, and the set baseline.
package_name	is the name of the package you want to build.

8. Issue the following commands to build the binary package.

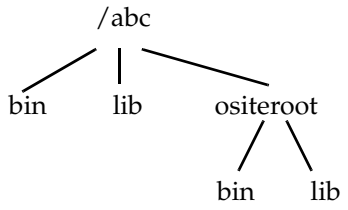
   ```
   bdir
   make_bin_pkg <package_name>
   ```

   If the build succeeds, the new executables and shared objects are placed in the appropriate subdirectory under the prefix directory.

   If the build fails, issue a make_clean command from the directory holding the package, and repeat steps 6-8.

9. Copy the executables and shared objects from the build directory to your production area, preferably under a separate site directory. This way the additional development is kept separate from the original version of the product.

If the production directory is /abc and the site directory is *ositeroot*, the following structure should be created:



10. Update $PATH with the new directories.

    - ositeroot/bin
    - ositeroot/lib/LD_LIBRARY_PATH (for Solaris and OSF platforms)
    - ositeroot/lib/SHLIB_PATH (for HP platforms)
    - ositeroot/lib/LIBPATH (for AIX platforms)

11. If you have to change, add, or delete source files, repeat steps 6-10.

Note: If you need to make new executables, edit the appropriate Make-file.am file in the subdirectory under *srcdir*.

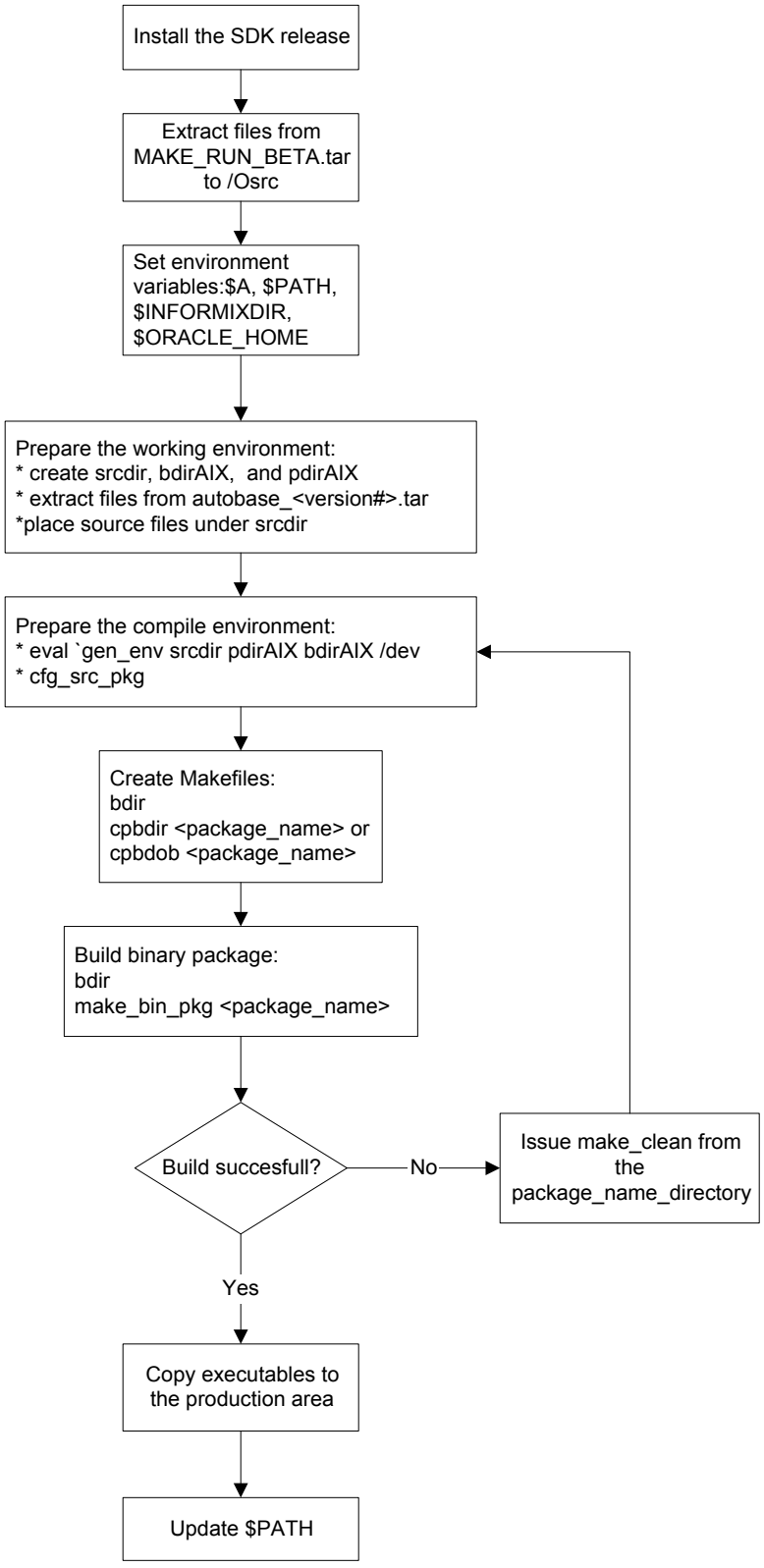


Figure 10:Compilation procedure

Part III: Technical Reference

Section 1: Source Package

1.1 Introduction

This section describes the different types of files in a package implementation under the source directory. The file categories are:

- Source files
- Documentation files
- Package specific configuration source files
- Generated configuration files

1.2 Source Files

A minimal package implementation has only source files. In addition to the traditional header and implementation files (C and C++, for instance), source files can also be *Makefile.am* files.

Each *Makefile.am* file describes the components of the sub-directory where the file sits in the package implementation. Therefore, there should be a separate *Makefile.am* file in every sub-directory in a package implementation.

The most common product types of package implementations are:

- Header files. These are header files the package makes public to the other packages. More on header files can be found later in the section.
- Library files. These are the binary library files that implement the package interface.
- Executable files. These are executable binary files.
- Sub-directories. This is not an actual product, rather it is the mechanism used to refer to the *Makefile.am* files in the sub-directories.

1.2.1 Header Files

There are four types of header files - public, local, implementational and generated. While the first three are source files, the last is generated during the configuration of each binary package.

1.2.1.1 Public Header Files

Public header files contain interface information (i.e. class / type definitions and function prototypes) of a package which is made public to other packages. This type of header file should be placed in the 'include' sub-directory of the package implementation.

To make header files public, their names must be declared in the *Makefile.am* file in the 'include' sub-directory with clause 'include_HEADERS = '. There should be at most one such clause in each *Makefile.am* file and blank characters (SPACE or TAB) to separate multiple files. Line continuation is also allowed.

For example, to declare 'efunds.h', 'syslg.h' and 'mb.h' as public, the following is specified in the 'include/Makefile.am'.

```
include_HEADERS =       \
    efunds.h  syslg.h   \
    mb.h
```

1.2.1.2 Local Header Files

Local header files contain interface information to be shared among sub-directories within a package implementation. Like their public counterpart, local header files should also be placed in the 'include' sub-directory. The difference is that they do not need to be declared in the *Makefile.am*.

1.2.1.3 Implementation Header Files

Implementational header files are referenced only within the sub-directory where they reside. They can be placed in any sub-directory except the ‘include’ sub-directory, since this does not contain implementation files.

1.2.1.4 Generated Header Files

Generated header files are one of the results of the configuration of each binary package. Currently, there is only one such header file, 'include/config.h', created in each binary package. The file is the means by which the configuration process communicates its findings to the source files, in the form of macros. Source files can use the definition of these macros for identification purposes, or to activate the sections of logic that are appropriate to the current configuration.

1.2.1.5 Header File Generation

Table 2: Header File Generation

Macro	Definition	Example
Package	Package name of the package implementation.	mbox_api
Version	The version number of the package implementation.	0.0.0
Target	A description of the platform where the configuration takes place.	alpha-dec-osf4.0d
HAVE_<object> where <object> can be name of a type, header file or function.	If defined, this indicates <object> is present in the current configuration.	#define HAVE_GETHOSTNAME 1 The function 'gethostname()' is present in this configuration.
NO_<object> where <object> can be name of a type, header file or function.	If defined, this indicates <object> is absent in the current configuration. This is the negation of the HAVE_ counterpart.	#define NO_UNION_WAIT 1 The union type 'wait' is absent from this configuration.
USE_<object> where <object> can be name of a type, header file or function.	If defined, this indicates <object> is preferred while there may be other alternatives.	#define USE_TERMIOS 1 The type 'terminos' is preferred in this configuration.

1.2.2 Library Files

Library files can be either dynamic-linked libraries (also known as shared libraries), or static-linked libraries. By default, libraries are created as dynamic-linked libraries, and unless you have a very good reason, the default should be used.

By convention, each library product should reside in a separate sub-directory. If the package implementation only produces one library, the sub-directory 'libsrc' can be used for the library. If there is more than one library, a descriptive sub-directory name should be chosen for each of them.

The name of each library should be prefixed 'libo', the standard designation for an eFunds library. The name should have the suffix '.la' instead of the platform specific extension (i.e. '.a', '.so' or '.sl'). The configuration process will determine the actual extension later.

The following information about the library must be declared in the *Makefile.am*:

- Name of the library, with lib_LTLIBRARIES =.
- List of implementation files (i.e. C, C++) that compose the library, with <library-file>_SOURCES where <library-file> is the name of the library with all '.' Substituted by '_'. For example, the `liboc.la` library becomes `liboc_la_SOURCES`.
- List of dependent libraries that the library requires, with <library-file>_LIBADD. The format of this list is very restricted. It can only consist of the following. Any other specifications (like 'a.o' or '/usr/lib/libm.a') are not allowed.

  -L<pathname>. Specify where the libraries specified with '-l' may reside.

  -l<library>. Specify the name of the library with the prefix 'lib' and file extension removed, and append the version at the desired level of specification. For example, for interface #0 of library *liboshtab.la*, you should specify *-loshtab-0*.

  Symbol name of one of the commonly used platform-specific libraries. The following are defined:

  $(POSIX_LIBS). The POSIX library.

  $(CPP_LIBS). The C++ support library.

  For example:

  ```
  liboedict_la_LIBADD = -lodict-0 -lotrace-0 $(CPP_LIBS)
  ```

- Linking flags, with *<library-file>_LDFLAGS*. This gives the explicit declaration of the library version number. For example, to declare the version of 1.2.3 for library *liboshtab.la*, specify *liboshtab_la_LDFLAGS = -release 1.2.3*.

With few exceptions, all dynamic-linked libraries should be made self-sufficient; that is, all dependent libraries should be specified in the *Makefile.am*.

For example, the *Makefile.am* for library *liboedict.la* in the package implementation *shtab_api-0.0.0* contains the following:

```
lib_LTLIBRARIES = liboedict.la
liboedict_la_SOURCES =  ist_edict.c
liboedict_la_LIBADD = -lodict1-0 -lotrace-0 $(CPP_LIBS)
liboedict_la_LDFLAGS = -release 0.0.0
```

1.2.3 Executable Files

By convention, executable and library products should not reside in the same sub-directory. Multiple executable products can, however, share the same sub-directory. A popular name for a sub-directory for executables is *bin*.

The following information about executables must be declared in the *Makefile.am*:

- Name of executables, with *bin_PROGRAMS* =. There should be one such declaration per *Makefile.am*. All executables are listed after the equal sign.

- List of implementation files (i.e. C, C++) that compose each executable, with *<executable-file>_SOURCES* where '<executable-file>' is the name of the executable.
- List of dependent libraries that the executable requires, with *<executable-file>_LDADD*. The syntax is the same as that of *<library-file>_LIBADD* for libraries. If some dependent libraries are required by all executables in the *Makefile.am*, they can be specified in 'LDADD' and '<executable-file>_LDADD' only contains those that are specific to each executable.

For example, the *Makefile.am* for executable 'blob', 'dec' and 'dec1' in package implementation 'oc_lib-0.0.0' contains the following:

```
bin_PROGRAMS = blob dec dec1
LDADD = -loc-0
blob_SOURCES = blob.cpp
dec_SOURCES = dec.cpp
dec1_SOURCES = dec1.cpp
```

There are additional specifications to make in the *Makefile.am* if the function 'main' of some of your executables does not reside in a C++ file (i.e. it resides in a C file). In this case, you need to create a C++ file and add the name of this file to the source list of these executables. The C++ file should contain:

```
static char _SCCSid[] = "@(#) %M% %I% %E% %U%";
#if defined( HP_UX )
extern "C" {
extern int efunds_C_main(int argc, char *argv[]);
}
int main(int argc, char *argv[])
{
    return efunds_C_main(argc, argv);
}
#endif
```

The reason why this file is needed is to ensure that the C++ compiler performs the linking, instead of the C compiler. The C++ compiler is needed since some of the low-level libraries contain C++ code and the C++ compiler is required to resolve their C++ references.

1.2.4 Sub-directories

The list of sub-directory names to be referred to are declared with *SUBDIRS* =.

For example, to refer to sub-directory 'include', 'libsrc' and 'src':

```
SUBDIRS = include libsrc src
```

1.3 Documentation Files

Each package implementation can have a number of files that store the documentation about the package. The following standard documentation files are

defined. If they are not present in a package implementation, they will be created (as empty files) during the configuration process.

AUTHORS	Information about the author(s) of the package implementation.
COPYING	Information about the copyright of the package implementation.
ChangeLog	History of changes in the package implementation.
INSTALL	Installation instructions for the package implementation.
NEWS	Upcoming features and trends for future package implementations.
NOTES	For information that does not fit in the other documents.
README	Highest level of documentation. May act as the table of contents for the other documents.

1.4 Package-Specific Configuration Source Files

These files are fragments of shell-scripts or meta-shell-scripts – m4 scripts that generate shell-scripts. The command *cfg_src_pkg* combines these files with their counterparts ('mid_configure.in' and 'aclocal.m4') in the autobase package to generate the configuration shell-script.

There are three files in this category:

top_configure.in	This shell-script fragment goes before the *mid_configure.in* in the resultant *configure.in* file.
bot_configure.in	This shell-script fragment goes before the *mid_configure.in* in the resultant *configure.in* file.
user_aclocal.m4	This meta-shell-script fragment goes after the *aclocal.m4* in the resultant *aclocal.m4* file in the package implementation. See Generated Configuration Files on page 37.

Most developers do not need to concern themselves with these files and no further detail about them will be supplied. Working with these files requires knowledge of GNU tool *autoconf*. For people who are familiar with this tool, the above description should be sufficient.

For more information about gnu tools, see the web page 'http://www.gnu.org/manual/manual.html'.

1.5 Generated Configuration Files

These are files that are generated by *cfg_src_pkg*.

configure.in	a combination of the *top_configure.in*, *mid_configure.in* and *bot_configure.in* files generates this.
aclocal.m4	a combination of the *aclocal.m4* file in *autobase* and the one in the package implementation generates this.
configure	Command *autoconf* generates this from *configure.in* and *aclocal.m4*. This is the configuration shell-script that is executed by *cfg_bin_pkg* to generate binary packages.
configure_fast	the command *cfg_src_pkg* generates this from *configure*. It differs from *configure* in that it skips the configuration tests if it finds the result of a previous configuration (stored in the configure.set file) of another similar package. See page 38 for *configure.set*. This is the same idea as the caching mechanism of *configure*, but extended to cover multiple packages. Command *cfg_src_pkg* determines that two packages are similar if they both have empty *top_configure.in* and *bot_configure.in* files.
scripts/*	A collection of shell-scripts copied from package *autobase*. Among these scripts are the gnu libtool tools that provide platform independent compiler/linker abstraction.
acconfig.h	This file is copied from package *autobase*. This is used to generate the *include/config.h.in* file below.
include/config.h.in	This generates the *include/config.h* file in the binary package.
include/stamp.h.in	This file acts as the timestamp of the configuration process.

These files are of no concern to most developers, and no further information needs to be supplied.

Section 2: Binary Packages

2.1 Introduction

This section describes the different types of files in a package under the binary directory. The categories of files are:

- Configuration data/log files
- Script files
- Package sub-directories
- Makefiles
- Product files

2.2 Configuration Data/Log Files

The configuration data/log files record the progress of the configuration process of the binary package.

config.log	This is an autoconf log file that contains the output of the tests performed during the configuration. The file can be examined for debugging purpose.
config.cache	This is an autoconf state file. This file stores the result of the tests performed during the configuration. The original idea is that if the result of a test can be found in the cache, the test can be skipped.
config.status	This is an autoconf file. This shell-script contains the procedures that generate the Makefiles. The original intent was for this shell-script to be executed to regenerate the Makefiles after the binary package is configured. The command *cfg_bin_pkg* provides the same functionality over a larger scope, since it covers multiple packages. It also provides a more consist user interface at roughly the same efficiency.
configure.set	This is a shell-script containing procedures to restore the state of the configuration process from previous configurations, not only that of the test results. Package-specific information is not included in the file. Consequently, this file can be shared among similar packages (re. the definition of “similarity” in the subsection on Generated configuration files. See Generated Configuration Files on page 37). This file is

	created and read by *configure_fast* under the control of *cfg_bin_pkg*.
cfg_bin_pkg	This command hides the detail regarding how these files are manipulated. Most developers do not need to be concerned with these files.

2.3 Script files

There is only one script file - *libtool*. The shell-script is the platform independent abstraction of the compiler and linker. The Makefiles call *libtool* in place of the compiler and linker.

2.4 Package Sub-directories

These mirror the sub-directories in the source packages. Each of these sub-directories contains a Makefile and product files generated from the source files in the corresponding sub-directories in the source directory.

2.5 Makefiles

The Makefiles are generated from the *Makefile.am* in the corresponding sub-directory under the source directory.

2.6 Product Files

These refer to the object files ('.o'), logical library files ('.la' or '.lai'), physical library files ('.a', '.so' or '.sl') and executable files that result from the compilation and linking of the source files. Note that the physical library files are located in the sub-directory '.libs' and logical library files are data files that are managed by the GNU command *libtool*.

Section 3: Package Database

3.1 Introduction

The source files that are required to construct a development environment are stored in an archived and versioned database file. Questions regarding which package implementations to retrieve from the archive can be answered by the package database.

The package database stores the summary of all package implementations, and it can answer questions regarding dependencies among package implementations. With the database, a developer can determine which packages are required to build specific library or executable files, and obtain only these needed packages from the archive. Without a database, the developer will be forced to guess and probably waste time retrieving unnecessary packages.

3.2 Relationships in the Package Database

The package database stores two types of relationships for each package implementation:

- Container relationship. The elements (i.e. header, library and executable files) that each package implementation contains or builds.
- Dependency relationship. The elements (i.e. header, library and executable files) that each package implementation depends on.

In other words, the package database stores everything that has been mentioned previously (See Package Definition on page 5), including the names and version of the package implementations, library and executable files, and their relations.

Note 1: The current version of the package database does not account for header files. The assumption is that the dependency information in header-files is always mirrored by their library-files counterpart.
Note 2: The current version of the package database does not account for executable files in the dependency relationships.

Like the package implementations, the package database is a file and is stored in the archive. Unlike the package implementations, however, there is only one file and the latest version is always available.

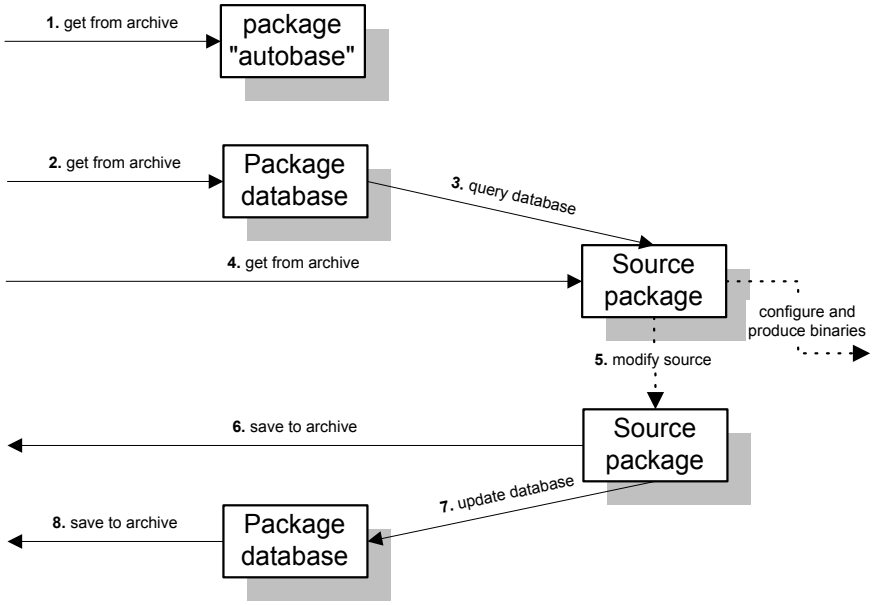


Figure 11:Package Database and Source Lifecycle

The diagram illustrates the activities of getting and saving source packages.

1. Install an implementation of package 'autobase' (the latest version usually). One important purpose is to install the package manipulation command scripts, including *calc_dependency*, to the public executable directory (by default, /usr/local/bin). The step is invoked by executing command *install_autobase* in the 'autobase' package. For example, the command

   ```
   ksh install_autobase --prefix=/Osrc/tools
   ```

   installs the command scripts to directory /Osrc/tools/bin. It is important to run *install_autobase* as a parameter of *ksh* since the former has not been given execution permission. Typically, /Osrc/tools/bin (or another directory where the scripts are installed) should be appended to the environment variable $PATH.

2. Get the package database file from the archive. This step is not necessary if you know exactly which package implementations are needed. Skip to step 3 if this is the case.

3. Query the package database regarding which package implementations are needed to build the set of executable and library files that you need. This is done with the utility *calc_dependency*, which acts on the package database obtained in step 1. For example, the command

   ```
   calc_dependency mb mbtsk
   ```

returns *header-0.0.0, mbox_api-0.0.0, mbox_util-0.0.0, misc_api-0.0.0, oc_lib-0.0.0, os_lib-0.0.0, static_api-0.0.0, tasks_api-0.0.0, tasks_util-0.0.0* and *trace_api-0.0.0* - the list of package implementations required to build the executables 'mb' and 'mbtsk'. Where there is more than one combination of package implementations, the tool will prompt for your choice. More details about *calc_dependency* are given later in this manual. You also need an implementation of the special package 'autobase'. This package contains code and data to configure all other packages. This package is read-only and, unless you have a special reason not to, you should get the latest version.

4. Retrieve the package implementations from the archive. At the end of this step, all the package implementations should be staged under the source directory and be ready for configuration and production of binary files.

5. Perform modification to the source files in the packages. Remember that any modification to a package will result in the incrementation of its version number. In other words, you never modify an existing package implementation, you only create new ones.

6. Save the new package implementations to the archive.

7. Capture the information of the new package implementations to the package database. This is done with the *calc_dependency* script. For example, the command

   ```
   calc_dependency -update=mbox_util-0.1.2,tasks_util-0.2.4
   ```

   captures the information of package implementations *mbox_util-0.1.2* and *tasks_util-0.2.4* to the package database.

8. Save the updated package database to the archive.

Section 4: Package Requirements

4.1 Requirements

The tools defined in the tables below are required to manage the packages. They must be reachable from the executable search path.

Table 3: Tools Requirements

Tools	Description	Location
`sh` and `ksh`	Shell interpreters (these can be links to the bash shell if the other shells are unavailable)	`/bin`
GNU tools: m4 (Required by autoconf and automake.)	Macro preprocessor from GNU m4 package version 1.4 or later. [a]	`/Osrc/local/bin/$A` (or another directory specified during the installation of m4)
GNU tools: autoconf / autoheader	From GNU autoconf package version 2.13 or later.	`/Osrc/local/bin/$A` (or another directory specified during the installation of auto-conf)
GNU tools: automake	From GNU automake package version 1.4 or later. *	`/Osrc/local/bin/$A` (or another directory specified during the installation of auto-make)
GNU make	Makefile interpreter from GNU make package version 3.74 or later *	`/efunds/gmake/$A` (or another directory specified during the installation of make)
perl	Perl interpreter 5.005_002 or later *	`/bin/perl` (`/tmp/perl`) (`/bin/perl` can be a link to the actual location where perl is installed)

Table 3: Tools Requirements (Continued)

Tools	Description	Location
gen_env calc_dependency cfg_src_pkg cfg_bin_pkg make_bin_pkg save_src_pkg restore_src_pkg chk_pkg_env pkgmap pkghelp	eFunds packaging tool set. (These tools come with the special package *autobase* and are installed with the script *install_autobase* - also from autobase.)	`/Osrc/tools/bin` (or another directory specified with the option `'-prefix=<DIR>'` when running *install_autobase*)
Appropiate compiler and linker.		Platform specific.

a. All the GNU software can be downloaded from www.gnu.org and the perl interpreter from www.cpan.org .

The environment variables listed below must be set before you commence building packages.

Table 4: Environment Variables

Environment Variable	Description
$DEFAULT_SRCDIR	The path to the source directory. If this is not defined, `/Osrc/src` is assumed.
$DEFAULT_PREFIX	The path of to the prefix directory. If this is not defined, it is assumed to be the same as the build directory. `$DEFAULT_PREFIX/bin` should be in the executable search path. `$DEFAULT_PREFIX/lib` should be in the DLL search path.
$ORACLE_HOME or $INFORMIXDIR	The path to the database vendor directory. One of these must be set when building the dbm_api package. `$ORACLE_HOME/lib` (or `$INFORMIXDIR/lib`) should be in the DLL search path.

The table below describes the tools provided by eFunds Corporation to assist in the management and use of the packages build environment.

Table 5: eFunds Corporation Packaging Tools

Tool	Description
gen_env	Given the source-dir and prefix-dir, generate the shell commands to set the `$DEFAULT_SRCDIR`, `$DEFAULT_PREFIX`, executable and DLL search path.
chk_pkg_env	This tool examines your environment and reports any errors and warnings.
opc	Package source control tool. See Package Source Control – OPC on page 65 for *opc*.
calc_dependency	The tool that manages the package dependency database. See calc_dependency on page 47.
cfg_src_pkg cfg_bin_pkg make_bin_pkg	The eFunds Corporation package build toolset. These tools hide most of the detail of autoconf and automake, and provide a more user-friendly interface. See cfg_src_pkg (csp) on page 51, See cfg_bin_pkg (cbp) on page 54, See make_bin_pkg (mbp) on page 57.
save_src_pkg restore_src_pkg	These tools archive source packages in gzip format. See save_src_pkg on page 61, and See restore_src_pkg on page 62.
pkgmap	The tool that manages the library symbol database. See pkgmap on page 63.
pkghelp	An interactive online help script which provides information about the Packages implementation and its inventory of tools.

Section 5: Package Command Specifications

5.1 Introduction

This section provides a detailed description of the commands used in the new environment, including a full description of command options and examples of use. The commands are:

- calc_dependency
- cfg_src_pkg
- cfg_bin_pkg
- make_bin_pkg
- install_autobase
- save_src_pkg
- restore_src_pkg
- pkgmap
- opc

5.1.1 Command Aliases

During the setup of the packages environment, some of these commands are aliased. These aliases are:

```
alias bdir="cd \$DEFAULT_BDIR" ;
alias pdir="cd \$DEFAULT_PREFIX" ;
alias sdir="cd \$DEFAULT_SRCDIR" ;
alias bldir="cd \$DEFAULT_BASELINEDIR" ;
alias repo="cd /Osrc/repository" ;
alias csp="cfg_src_pkg --m=s";
alias cbp="cfg_bin_pkg --m=s";
alias cbpd="cfg_bin_pkg --m=s ---enable-symbols" ;
alias cbpdo="cfg_bin_pkg --m=s ---enable-symbols ---with-ora-
cle=\$ORACLE_HOME" ;
alias cbpdi="cfg_bin_pkg --m=s ---enable-symbols ---with-infor-
mix=\$INFORMIXDIR";
alias cbpo="cfg_bin_pkg --m=s ---with-oracle=\$ORACLE_HOME" ;
alias cbpi="cfg_bin_pkg --m=s ---with-informix=\$INFORMIXDIR" ;
alias mbp=make_bin_pkg ;
alias cald=calc_dependency ;
alias x=". cdpkg" ;
```

5.2 calc_dependency

The *calc_dependency* command queries and updates the package database. It can calculate the 'dependency closure' for a combination of packages, libraries and executables; that is, the collection of package implementations that are required to build this combination. This collection also includes the package implementations that are required indirectly.

Syntax

```
calc_dependency [OPTION] ... [PACKAGE/LIBRARY/EXECUTABLE] ...
```

Options

Option	Description
--build[=CMD]	Perform build operation on the packages in the dependency closure. The least dependent packages are operated upon first. The algorithm spawns processes for operations that can be executed in parallel. CMD must be one of the following: <ABSENT> if CMD is absent, just display the packages in the dependency closure in the reverse dependency order (i.e. the least dependent ones are listed first) showplan display the concurrent build plan of the packages in the dependency closure test perform a serial build simulation on the packages in the dependency closure makeinstall perform concurrent 'make install' to the packages in the dependency closure. <OTHER> execute the specified build command. The versioned package name is passed to the command as the first argument.
--donotack	Do not prompt for input. When an ambiguous element is encountered, the command fails.
--file=FILE	Specify the pathname of the package information database. If this is not specified, it is assumed to be file *pkg.info* under the directory defined in environment variable $DEFAULT_SRCDIR if defined. Otherwise, it is assumed to be /Osrc/src/pkg.info.
--help	Display help messages and exit.

Option	Description
--list=TABLE	List the specified internal table. TABLE must be one of the following: belong — table that maps elements to the packages they belong to; contain — table that maps packages to the elements they contain; depend — table that maps packages to the elements they depend on.
--update[=SCOPE]	Update the database with information extracted from Makefile.am files. SCOPE defines which Makefile.am files are to be considered. SCOPE is a comma separated list of packages/package-implementations. If SCOPE is absent, all Makefile.am files are considered.
--version	Print version information and exit.

Examples

1. Query for which package implementations are needed to build the required packages/elements.

   1.1 To determine the dependency closure of package '**mbox_util-0.0.0**' and a version of library '**liboc_na_factory**':

   ```
   calc_dependency mbox_util-0.0.0 liboc_na_factory
   ```

   Output:

   ```
   header-0.0.0
   mbox_api-0.0.0
   mbox_util-0.0.0
   misc_api-0.0.0
   nodeaccess_lib-0.0.0
   oc_lib-0.0.0
   os_lib-0.0.0
   static_api-0.0.0
   tasks_api-0.0.0
   trace_api-0.0.0
   ```

   1.2 To determine, based on package database file '/tmp/pkg.info', the dependency closure of package '**mbox_util-0.0.0**' and library '**liboc_na_factory**':

   ```
   calc_dependency –file=/tmp/pkg.info mbox_util-0.0.0
   liboc_na_factory
   ```

   Output is same as above.

   1.3 Display serially the dependency closure of package '**mbox_util-0.0.0**' and library '**liboc_na_factory**' in the reverse dependency order:

   ```
   calc_dependency –build mbox_util-0.0.0 liboc_na_factory
   ```

Output (header is built first, while mbox_util is built last):

```
header-0.0.0
oc_lib-0.0.0
static_api-0.0.0
nodeaccess_lib-0.0.0
trace_api-0.0.0
os_lib-0.0.0
mbox_api-0.0.0
misc_api-0.0.0
tasks_api-0.0.0
mbox_util-0.0.0
```

1.4 Display concurrently the dependency closure of package '**mbox_util-0.0.0**' and library '**liboc_na_factory**' in the reverse dependency order:

```
calc_dependency –build=showplan mbox_util-0.0.0
liboc_na_factory
```

Output (header is built first, oc_lib and static_api next concurrently):

```
Building :  header-0.0.0
Building :  oc_lib-0.0.0 static_api-0.0.0
Building :  trace_api-0.0.0 nodeaccess_lib-0.0.0
Building :  os_lib-0.0.0
Building :  mbox_api-0.0.0 misc_api-0.0.0
Building :  tasks_api-0.0.0
Building :  mbox_util-0.0.0
```

2. Perform *make install* on packages implementations.

2.1 Build the dependency closure of package **mbox_util-0.0.0** and library **liboc_na_factory**:

```
calc_dependency –build=makeinstall mbox_util-0.0.0
liboc_na_factory
```

Output (header is built first, followed by oc_lib and static_api concurrently):

```
. . .
make[1]: Entering directory ` /Osrc/bdir/header/include'
. . .
```

3. Update package database with information from selected package implementations

3.1 Update the package database with information in package implementation **mbox_util-0.0.0** under the current directory:

```
calc_dependency --update=mbox_util-0.0.0
```

3.2 Update the package database with information in package implementation **mbox_util-0.0.0** and all implementations of package oc_lib under the current directory:

```
calc_dependency --update=mbox_util-0.0.0,oc_lib
```

3.3 Update the package database with information in all package implementations under the current directory:

```
calc_dependency --update
```

3.4 Update the package database **/tmp/pkg.info** with information in all package implementations under the current directory:

```
calc_dependency –update --file=/tmp/pkg.info
```

5.3 cfg_src_pkg (csp)

The **cfg_src_pkg** command, alias **csp**, updates and creates the configuration scripts of the specified packages and package-implementations in the current directory; that is, the source directory.

Syntax

```
cfg_src_pkg [OPTION] ... [PACKAGE] ...
```

PACKAGE can be the name of a package or package implementation. If **PACKAGE** is a package name, one of its implementations under the current directory will be chosen. There is no guarantee regarding which implementation will be chosen.

PACKAGE can also be the name of a library or executable, if option '**--dependent**' is specified.

If **PACKAGE** is not specified, the current directory will be assumed to be the directory of a particular package implementation, and the package name and version number are deduced from the directory name.

The order in which the package-implementations are processed is not important. In fact, ***cfg_src_pkg*** will attempt to spawn a separate process for each package-implementation and process all of them at once. This, however, creates a bundle to the machine running the process. Should this become a problem, specify the option —***mode=series*** to process the package-implementations in series.

Options:

Option	Description
--create=[yes|no]	allow or disallow the creation of new packages. By default the creation of new packages is disallowed.
--dependent	perform operation to the specified packages and those they depend on.
--dfile=DIR	specify the path name of the dependence database file. If DIR is omitted, it is assumed to be file *pkg.info* in the current directory. This also turns on *dependent*.
--dryrun	display the list of packages covered by the command and exit.
--help	display help message and exit
--mode=MODE	control the degree of concurrency. MODE must be one of the following:

Option	Description	
	parallel:	make with maximum concurrency. This is the default.
	series:	make packages in series.
--version	print version information and exit.	

Examples:

1. Update the configuration scripts for specific packages/package-implementations.

   1.1 Update the configuration scripts for package implementation **mbox_util-0.0.0**:

   ```
   cfg_src_pkg mbox_util-0.0.0
   ```

   Output (only the implementation mbox_util-0.0.0 is processed):

```
mbox_util === Configuring source package [ mbox_util ]
mbox_util === Using autobase directory [ autobase/autobase-0.1.9 ]
mbox_util === Configuring package [ mbox_util ] version [ 0.0.0 ]
mbox_util === Installing configure.in ...
mbox_util === Installing aclocal.m4 ...
mbox_util === Copying libtool/install script ...
mbox_util === Running automake ...
mbox_util === Running autconf ...
mbox_util === Generating configure_fast ...
mbox_util === This source-package is now updated with the latest
mbox_util     version of configuration scripts. Now, please run
mbox_util     'cfg_bin_pkg' and 'make_bin_pkg' on the affected
binary
mbox_util     packages.
=== All 1 Jobs ended
```

   1.2 Update the configuration scripts for package implementation **mbox_util-0.0.0** and one implementation of **oc_lib**:

   ```
   cfg_src_pkg mbox_util-0.0.0 oc_lib
   ```

   Output (Implementation oc_lib-0.0.0 has been chosen for oc_lib. Both implementations are processed in parallel.):

```
mbox_util === Configuring source package [ mbox_util ]
oc_lib === Configuring source package [ oc_lib ]
mbox_util === Using autobase directory [ autobase/autobase-0.1.9 ]
oc_lib === Using autobase directory [ autobase/autobase-0.1.9 ]
oc_lib === Configuring package [ oc_lib ] version [ 0.0.0 ]
oc_lib === Installing configure.in ...
mbox_util === Configuring package [ mbox_util ] version [ 0.0.0 ]
mbox_util === Installing configure.in ...
. . .
```

   1.3 Update serially the configuration scripts for package implementation **mbox_util-0.0.0** and one implementation of **oc_lib**:

```
cfg_src_pkg -mode=series mbox_util-0.0.0 oc_lib
```

Output (The implementations are processed one by one.):

```
mbox_util === Configuring source package [ mbox_util ]
mbox_util === Using autobase directory [ autobase/autobase-0.1.9 ]
mbox_util === Configuring package [ mbox_util ] version [ 0.0.0 ]
mbox_util === Installing configure.in ...
mbox_util === Installing aclocal.m4 ...
mbox_util === Copying libtool/install script ...
mbox_util === Running automake ...
mbox_util === Running autconf ...
mbox_util === Generating configure_fast ...
mbox_util === This source-package is now updated with the latest
mbox_util     version of configuration scripts. Now, please run
mbox_util     cfg_bin_pkg and make_bin_pkg on the affected binary
mbox_util     packages.
oc_lib === Configuring source package [ oc_lib ]
oc_lib === Using autobase directory [ autobase/autobase-0.1.9 ]
. . .
=== All 2 Jobs ended
```

2. Update the configuration scripts for the dependency closure of the specified combination of libraries/executables/packages/package-implementation.

   2.1 Update the configuration scripts for the dependency closure of library **liboc_na_factory** and executable **mb**:

   ```
   cfg_src_pkg -dependent liboc_na_factory mb
   ```

   2.2 Update the configuration scripts for the dependency closure of library **liboc_na_factory** and executable **mb** with explicit specification of the package database file /tmp/pkg.info:

   ```
   cfg_src_pkg -dependent -dfile=/tmp/pkg.info liboc_na_factory mb
   ```

3. Create a directory for a new package implementation and populate it with configuration scripts.

   3.1 Add a new implementation for new package **new_pkg**. The implementation will be **new_pkg-0.0.0**:

   ```
   cfg_src_pkg -create=yes new_pkg
   ```

   3.2 Add new package implementation **new_pkg-1.2.3**:

   ```
   cfg_src_pkg -create=yes new_pkg-1.2.3
   ```

5.4 cfg_bin_pkg (cbp)

The **cfg_bin_pkg** command, alias **cbp**, configures the specified packages and package implementations in the current directory; that is, the binary directory.

Syntax

```
cfg_bin_pkg [OPTION] ... [PACKAGE] ...
```

PACKAGE can be the name of a package or package implementation. If **PACKAGE** is a package name, one of its implementations under the current directory will be chosen. There is no guarantee regarding which implementation will be chosen.

PACKAGE can also be the name of a library or executable, if option **--dependent** is specified.

If **PACKAGE** is not specified, the current directory will be assumed to be the directory of a particular package implementation, and the package name and version number will deduced from the directory name.

The order in which the package-implementations are processed are not important. As a matter of fact, **cfg_bin_pkg** will attempt to spawn a separate process for each package-implementation and process all of them at once. This, however, creates a bundle to the machine running the process. Should this becomes a problem, specify option —**mode=series** to process the package-implementations one by one.

Options:

Options	Description
--dependent	perform operation to the specified packages and those they depend on
--dfile=DDIR	specify the path name of the dependence database file. If this option is not specified, it is assumed to be file pkg.info in the source directory (see --srcdir below). This also turns on --dependent.
--fast=[yes | no]	control if fast-configure should be used if applicable. Fast configuration is most effective when configuring a lot of packages in one batch.
--help	display help message and exit.
-mode=MODE	control the degree of concurrency. MODE must be one of: parallel — make with maximum concurrency. This is the default.

Options	Description
	series make one package after another.
--OPTION	pass as-is to configure script. This can be used to specify the database vendor and directory in the configuration of database packages.
--prefixdir=PDIR	specify the prefix directory. Prefix directory (GNU term) is the directory where the directory *include*, *lib* and *bin* are created. If this option is not specified, prefix directory is obtained from the environment variable $DEFAULT_PREFIX, if defined. Otherwise, which is the most usual situation, it is assumed to be the same as the binary directory; that is, the current directory.
--srcdir=SDIR	specify the source directory. If this option is not specified, the source directory is obtained from the environment variable $DEFAULT_SRCDIR, if defined. Otherwise, the source directory is assumed to be '/Osrc/src'.
--version	print version information and exit.

Examples:

1. Configure specific packages/package-implementations.

   1.1 Configure the package implementation **mbox_util-0.0.0**:

   ```
   cfg_bin_pkg mbox_util-0.0.0
   ```

   Output (mbox_util-0.0.0 is fast configured):

   ```
   Using fast-configure templates from pkg [ mbox_util ]
   Changed my mind - fast-configure templates are dropped since the
   configure script of pkg [ mbox_util ] has been modified
   Generating fast-configure template with pkg [ mbox_util ]
   Configuring package [ mbox_util-0.0.0 ] with srcdir [ /Osrc/src ]
   prefix [ /Osrc/bdirosf ]
   creating cache ./config.cache
   checking host system type... /Osrc/bdirosf
   . . .
   ```

   1.2 Configure one implementation of package **mbox_util**:

   ```
   cfg_bin_pkg mbox_util
   ```

   1.3 Configure serially the package implementation **mbox_util-0.0.0** and **mbox_api-0.0.0**:

   ```
   cfg_bin_pkg --mode=series mbox_util-0.0.0 mbox_api-0.0.0
   ```

   1.4 Configure the package implementation **mbox_util-0.0.0** in source directory **/tmp/sdir**:

```
cfg_bin_pkg --srcdir=/tmp/sdir mbox_util-0.0.0
```

2. Configure the dependency closure of combination of libraries/executables/packages/package-implementations.

   2.1 Configure the dependency closure of executable '**mb**' and library **liboc_na_factory**:

   ```
   cfg_bin_pkg -dependent mb liboc_na_factory
   ```

3. Configure specific packages/package-implementations with additional configuration options.

   3.1 Configure the informix database package **dbm_api**:

   ```
   cfg_bin_pkg ---with-informix=/apps/informix/ids7.24 dbm_api
   ```

   3.2 Configure the oracle database package with database implementation **dbm_api-3.0.0**:

   ```
   cfg_bin_pkg ---with-oracle=/apps/oracle/product/8.0.5 dbm_api-
   3.0.0
   ```

5.5 make_bin_pkg (mbp)

The **make_bin_pkg** command, alias **mbp**, performs make on the specified packages / package-implementations in the current directory; that is, the binary directory.

Syntax

```
make_bin_pkg [OPTION] ... [PACKAGE] ...
```

PACKAGE can be the name of a package or package implementation. If **PACKAGE** is a package name, one of its implementations under the current directory will be chosen. There is no guarantee regarding which implementation will be chosen.

PACKAGE can also be the name of a library or executable if option **--dependent** is specified.

If **PACKAGE** is not specified, the current directory is assumed to be the directory of a particular binary package, and the package name is deduced from the directory name.

Unlike ***cfg_src_pkg*** and ***cfg_bin_pkg***, the order in which the package implementations are processed is important. For this reason, ***make_bin_pkg*** processes each package serially by default. The ***calc_dependency*** command should be used instead of *make_bin_pkg* for better performance, while ***make_bin_pkg*** should be used if clarity is a priority.

Unless —**dependent** is specified, the order of the packages and package implementations specified in the command must be in the reverse order of dependency. The expanded list that results from option —**dependent** is always in reverse order of dependency.

Options:

Option	Description
--dependent	perform operation on the specificied packages and those they depend upon.
--dfile=DDIR	specify the path name of the dependence database file. If this option is not specified, it is assumed to be file pkg.info in the source directory (see --srcdir below). This also turns on --dependent.
--help	display help message and exit.
--mode=MODE	control the degree of concurrency. MODE must be one of the following:

Option	Description
	parallel make with maximum concurrency. This is the default series make one package after another.
--silent	perform the make silently.
--srcdir=DIR	specify the source directory. If this option is not specified, the source directory is obtained from the environment variable $DEFAULT_SRCDIR, if defined. Otherwise, the source directory is assumed to be /Osrc/src.
--version	print version information and exit.
--target=TARGET	specify the target to be made. If this option is not specified, target of 'install' is assumed. Target install is to: i Install public header files to prefix directory '**include**'. ii Compile the source files. iii Link libraries. iv Install libraries to prefix directory '**lib**'. v Link executables. vi install executables to prefix directory '**bin**'.

Examples:

1. Make specific packages/package-implementations.

   1.1 Make the package implementation **mbox_util-0.0.0**:

   ```
   make_bin_pkg mbox_util-0.0.0
   ```

   1.2 Configure one implementation of package **mbox_util**:

   ```
   make_bin_pkg mbox_util
   ```

2. Make the dependency closure of combination of libraries/executables/packages/package-implementations.

   2.1 Configure the dependency closure of executable **mb** and library **liboc_na_factory**:

   ```
   make_bin_pkg –dependent mb liboc_na_factory
   ```

3. Verify (i.e. make check) the correctness of specific packages/package-implementations.

   3.1 Verify the correctness of the package implementation **mbox_util-0.0.0**:

   ```
   make_bin_pkg –target=check mbox_util-0.0.0
   ```

3.2 Verify the correctness of the dependency closure of package implementation **mbox_util-0.0.0**:

```
make_bin_pkg  -target=test --dependent mbox_util-0.0.0
```

5.6 install_autobase

The **install_autobase** command installs the command scripts of the autobase package to the prefix directory.

If **ACTION** is absent the install action is assumed.

Syntax

```
install_autobase [OPTION] ... [ACTION]
```

Options:

Option	Description
--prefix=PREFIX	install script files in PREFIX/bin. By default, PREFIX is **/usr/local**.
--help	display help message and exit
--version	print version information and exit

ACTION must be one of the following:

install	install the script files
uninstall	uninstall the script files

Examples:

1. Install the script files.

   1.1 Install the script files to /usr/local/bin:

   ```
   Install_autobase
   ```

   1.2 Install the script files to /Osrc/tools/bin:

   ```
   Install_autobase –prefix=/Osrc/tools
   ```

2. Uninstall the script files.

   2.1 Uninstall the script files from /usr/local/bin:

   ```
   Install_autobase uninstall
   ```

   2.2 Uninstall the script files from /Osrc/tools/bin:

   ```
   Install_autobase –prefix=/Osrc/tools uninstall
   ```

5.7 save_src_pkg

The *save_src_pkg* archives the specified package version to the archive directory under the source directory. This command should be executed in the source directory.

Syntax

```
save_src_pkg [OPTION] ... [PACKAGE-VERSION] ...
```

Options:

Option	Description
dependent	performs the operation on the specified packages and those packages they depend upon.
dfile=dir	Specifies the pathname of the dependency database file. If omitted, this is assumed to be the `pkg.info` file in the current directory. This option also turns on the *dependent* option.
help	Displays the help message and exits.
mode=MODE	Controls the degree of concurrency. MODE must be one of the following: *parallel* - make with maximum concurrency *series* - make packages serially; i.e., one after the other.
overwrite	Overwites the archive file if it exists.
remove	Removes the source package after saving.
version	Prints the version information.

5.8 restore_src_pkg

Restore the specified package version from the archive directory under the source directory. This command should be executed in the source directory.

Syntax

```
restore_src_pkg [OPTION] ... [PACKAGE-VERSION] ...
```

Options:

Option	Description
dependent	performs the operation on the specified packages and those packages they depend upon.
dfile=dir	Specifies the pathname of the dependency database file. If omitted, this is assumed to be the `pkg.info` file in the current directory. This option also turns on the *dependent* option.
help	Displays the help message and exits.
mode=MODE	Controls the degree of concurrency. MODE must be one of the following: *parallel* - make with maximum concurrency *series* - make packages serially; i.e., one after the other.
overwrite	Overwites the archive file if it exists.
remove	Removes the source package after saving.
version	Prints the version information.

5.9 pkgmap

eFunds library symbol browser. It builds symbol database by extracting symbol information in library files. With the symbol database, it answers queries like 'which library defines these symbols ?' or 'what symbols are defined in this library ?'.

If no COMMAND is specified, the program will run in interactive mode. The program prompts user for COMMAND during interactive mode until 'exit' or 'quit' is specified.

Syntax

```
pkgmap [OPTIONS] [COMMAND]
```

Options

Option	Description
file=<DATABASE-FILE>	This causes the program to read from an alternative database file. The default is a file called *symbol.info* in the current directory.
dryrun	This skips the write instruction to the database file.

Commands

Command	Description
add [LIBRARY-FILE] ...	Add the symbols in the specified library-files to the database. If no library-files are specified, all library files in the current directory are processed.
sym [SYMBOL-NAME|CLASS-NAME] ...	Return the library name(s) where the specified symbol/class name(s) are defined. If no name is specified, the information for all symbols is returned.
lib [LIBRARY-NAME] ...	Return the symbol/class names that are defined in the specified library(s). If no name is specified, the information of all libraries is returned.
Exit / Quit	Exit the program.

Note: the symbols that are reported by this tool are not necessarily part of the public interface of eFunds Corporation software. These symbols are used at your own risk.

5.10 Package Source Control – OPC

The *opc* program manages the life-cycles of packages. It is the equivalent of SCCS at the package level. The sub-command names intentionally mimic those of SCCS to indicate that they provide similar functionality.

When manipulating individual files, SCCS should be used to manage their life-cycles.

When running in package or a package-implementation context, *opc* must be run in the root directory of the source packages.

Syntax

```
opc [ PKGIMPL-CMD | PKG-CMD | DIR-CMD | FILE-CMD ]
```

Commands

PKGIMPL-CMD:

These are the package-implementation level commands.

Command	Description
create [pkg-impl] ...	Given a list of package implementations, for each implementation, build an SCCS-package from the SCCS directories.
get [-d<dir>|--dst-dir=<dir>] [pkg-impl] ...	Given a list of package implementations, instantiate the specified implementations under the current directory. If '-d' or '--dstdir=' are specified, the implementations will be instantiated in <dir> instead.
unget [-d<dir>|--dst-dir=<dir>] [pkg-impl] ...	Given a list of package implementations, remove the specified implementations from the current directory. If '-d' or '--dstdir=' are specified, the implementations will be removed from <dir> instead.
edit [pkg-impl] ...	Given a list of package implementations, check out the specified implementations.
unedit [pkg-impl] ...	Given a list of package implementations, reverse the check out of the specified implementations.
delget [-y<comment>|--y=<comment>] [pkg-impl] ...	Given a list of package implementations, check in the specified implementations. The '-y' or '--y' specifies a comment to be attached to the new package implementation.

Command	Description
diffs [-r<pkg-ver>|--r=<pkg-ver>] [pkg-impl] ...	Given a list of package implementations, for each implementation, display the differences (in terms of files and their versions) between what is currently under the implementation directory and the captured file list. By default, the operation compare against the captured file list of the directory. If '-r' or '--r=' are specified, that of the specified implementation is used instead.
list [pkg-impl] ...	Given a list of package implementations, list the name and version of all SCCS'ed files that have been captured in the pkg-version file.
version [-u|--unsafe] [pkg-impl] ...	Given a list of package implementation, display the name and version of all SCCS'ed files currently in the directories of the specified package implementations.
scan [-t<spec>|--type=<spec>] [pkg-impl] ...	Given a list of package implementations, perform 'scandir' on the directories of the specified package implementations.
examine [-i|--ignore-all] [-l<path>|--log=<path>] [pkg-impl] ...	Given a list of package implementation, perform 'examinedir' on the directories of the specified package implementations.

PKG-CMD:

These are the package level commands.

Command	Description
link [-d<dir>|--dst-dir=<dir>] [pkg] ...	Given a list of package names and a destination directory, make the SCCS-package of the specified packages, that are accessible from the current directory, also available from the destination directory specified in <dir>.
info [pkg] ...	Given a list of package names, display the names of all available implementations and those that have been checked out.

DIR-CMD:

These are the directory level commands.

Command	Description
versiondir [-u|--unsafe] [dir] ...	Given a list of directories, display the name and version of all contained SCCS'ed files.

Command	Description
scandir [-t<spec>|--type=<spec>] [dir] ...	Given a list of directories, for each directory, categorize the files as one of four following types and then display them: - source file, (must be SCCS'ed) - working file, (no need to be SCCS'ed) - garbage file, (no need to be SCCS'ed and should be deleted) - unclassified file By default, only source and unclassified files are displayed. If '-t' or '--type=' are specified, <spec> determines which type(s) of files are to be displayed. <spec> is a combination of 's', 'w', 'g', 'u' or 'a' (for all).
examinedir [-i|--ignore-all] [-l<path>|--log=<path>] [dir] ...	Given a list of directories, for each directory, first, categorize the files, as described in the *scandir* command above. For each source or unclassified file that is either checked-out or not yet SCCS'ed, prompt the user for action to deal with them (i.e. check-in, SCCS, delete or ignore). If '-i' or '--ignoreall' is specified, instead of prompting the user for action, write the file name to the log file. If '-l' or '--l=' is specified, instead of write to log file 'pkg.log' in the current directory, write to file specified in <path> instead.

FILE-CMD:

These are the file level commands.

Command	Description
versionfile [-u|--unsafe] [file] ...	Given a list of files, display its name and version.
editfile [file] ...	Given a list of files, edit the current version of each file (instead of its latest version, as is the case with *sccs edit*).
uneditfile [file] ...	Given a list of files, unedit each file and returns it to its original version (instead of its latest version, as is the case with *sccs unedit*)

Notes

The version of a file is determined according to the following schema.

First, only read-only files are examined. Writable files will cause the process to terminate.

By default, a file is matched through 'sccs diffs' against each stored file version to determine its actual version. The in-file sccs-id is also extracted and compared with the found file version. If they don't match, a warning message is generated.

If '-u' or '--unsafe' is specified, 'sccs diffs' comparison is not performed if the in-file sccs-id can be found. This is 'unsafe' in the sense that the result may not be reliable, although it may be faster.

Appendix A:

Graphics

a.1 Overview

This Appendix presents graphical representations of the directory structures for the packages build environments. These are to be used in conjunction with the examples in Part II of this manual.

a.2 Source Repository

The graphic below describes the respective work areas within the packages source repository.

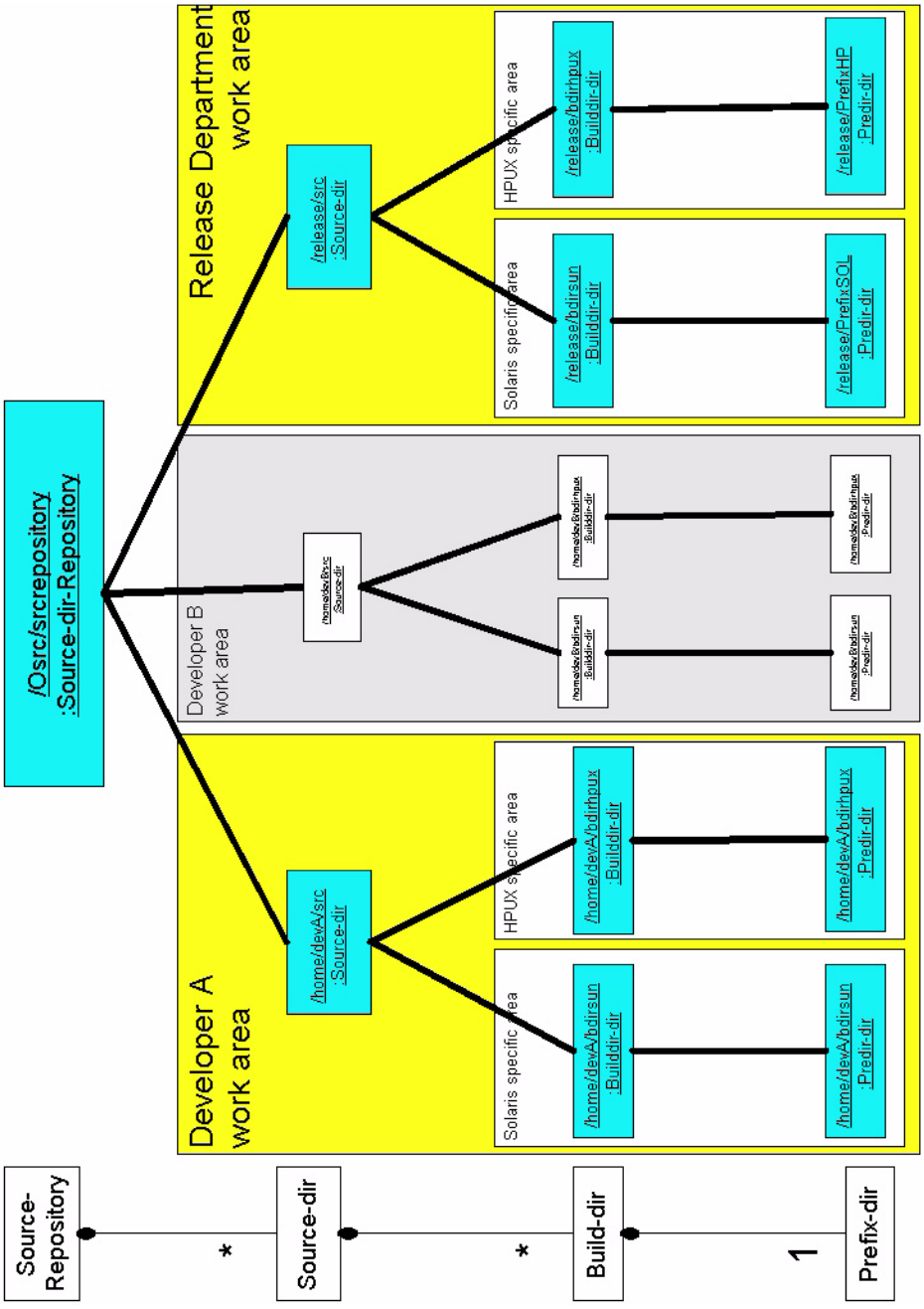


Figure 12:Source Repository

a.3 Source Directory of Package header-0.0.0

This graphic illustrates the structure of the package header-0.0.0 directly after staging but before execution of the cfg_src_pkg command. See Example 1: Build liboc Library on page 14, Step 4.

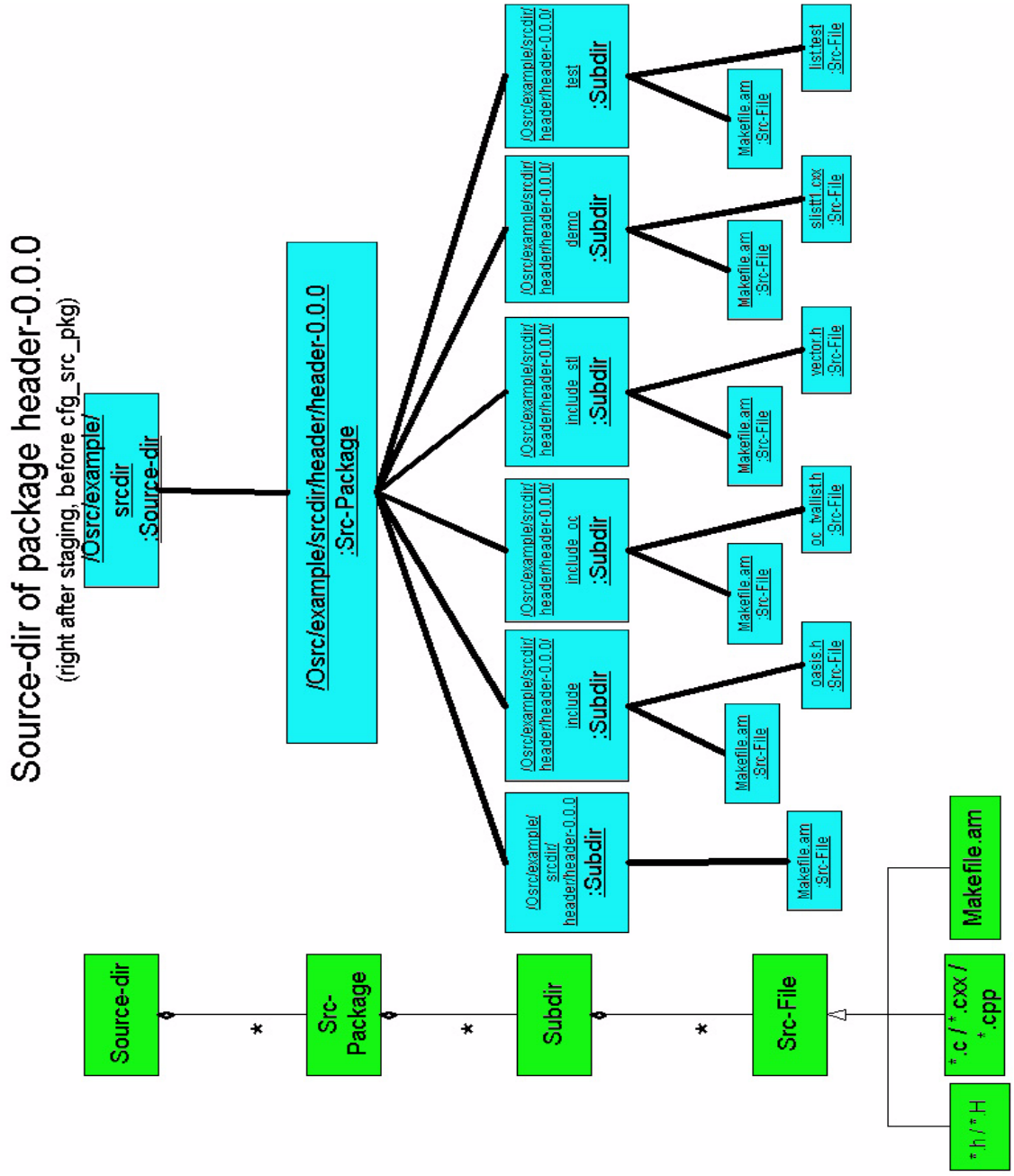


Figure 13:Source Directory of Package header-0.0.0

a.4 Source Directory of Package oc_lib-0.1.0

This graphic illustrates the source directory structure of package oc_lib-0.1.0 directly after staging and before execution of the cfg_src_pkg command. See Example 1: Build liboc Library on page 14 Step 4.

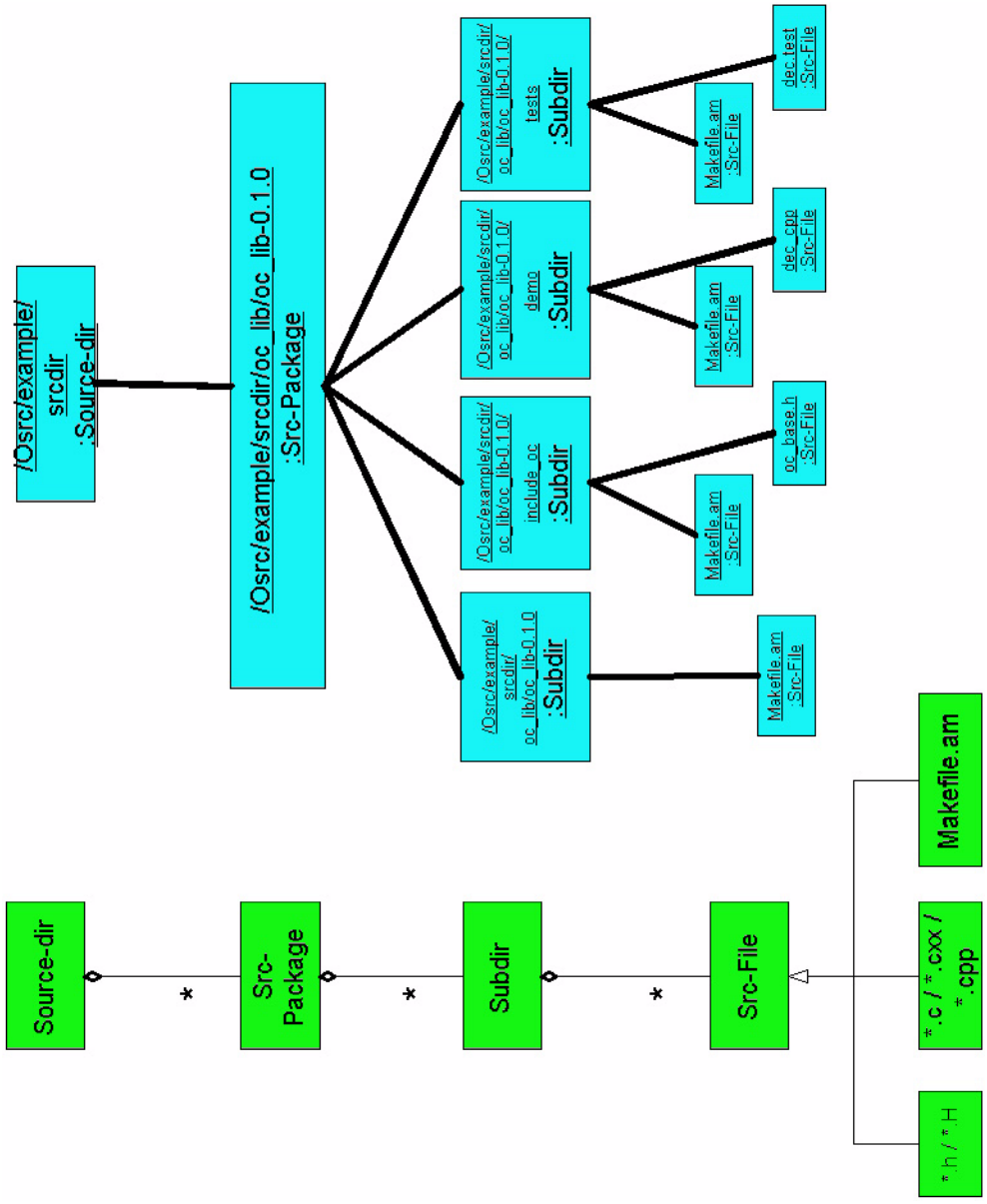


Figure 14:Source Directory of Package oc_lib-0.1.0

a.5 Configure Source and Binary Packages

The diagram below illustrates the environment that is constructed following steps 6 and 7 in the first example given in Part II of this manual. See Example 1: Build liboc Library on page 14.

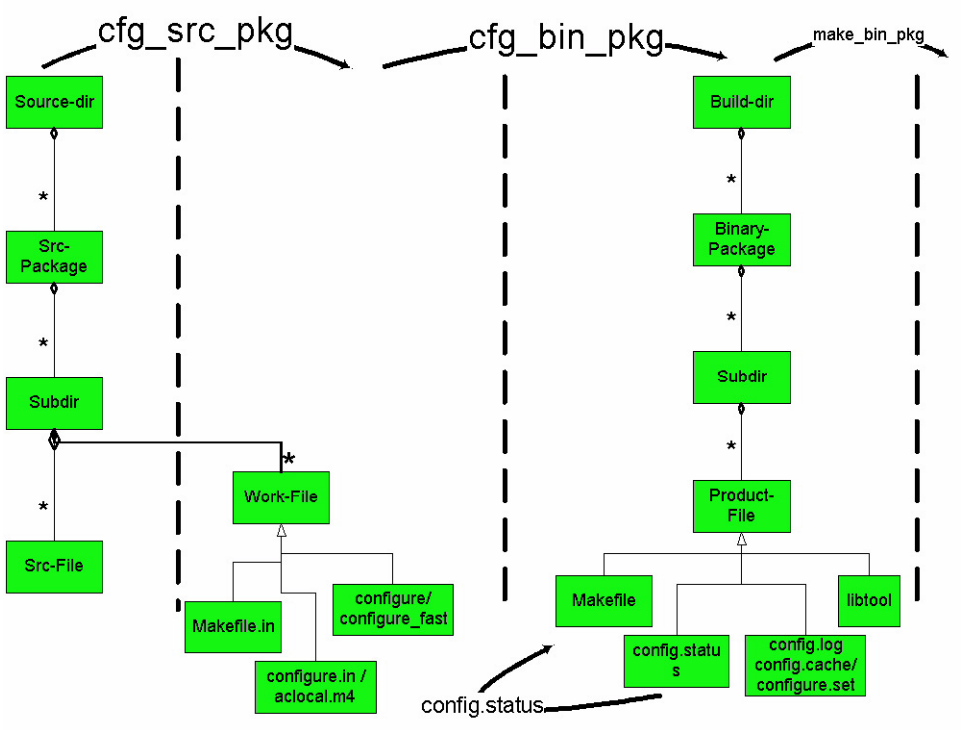


Figure 15:Source and Binary Package Directory Structure

a.6 Make Binary Packages

This graphic illustrates the directory structure created following Step 8 of the first example given in Part II of this manual. See Example 1: Build liboc Library on page 14.

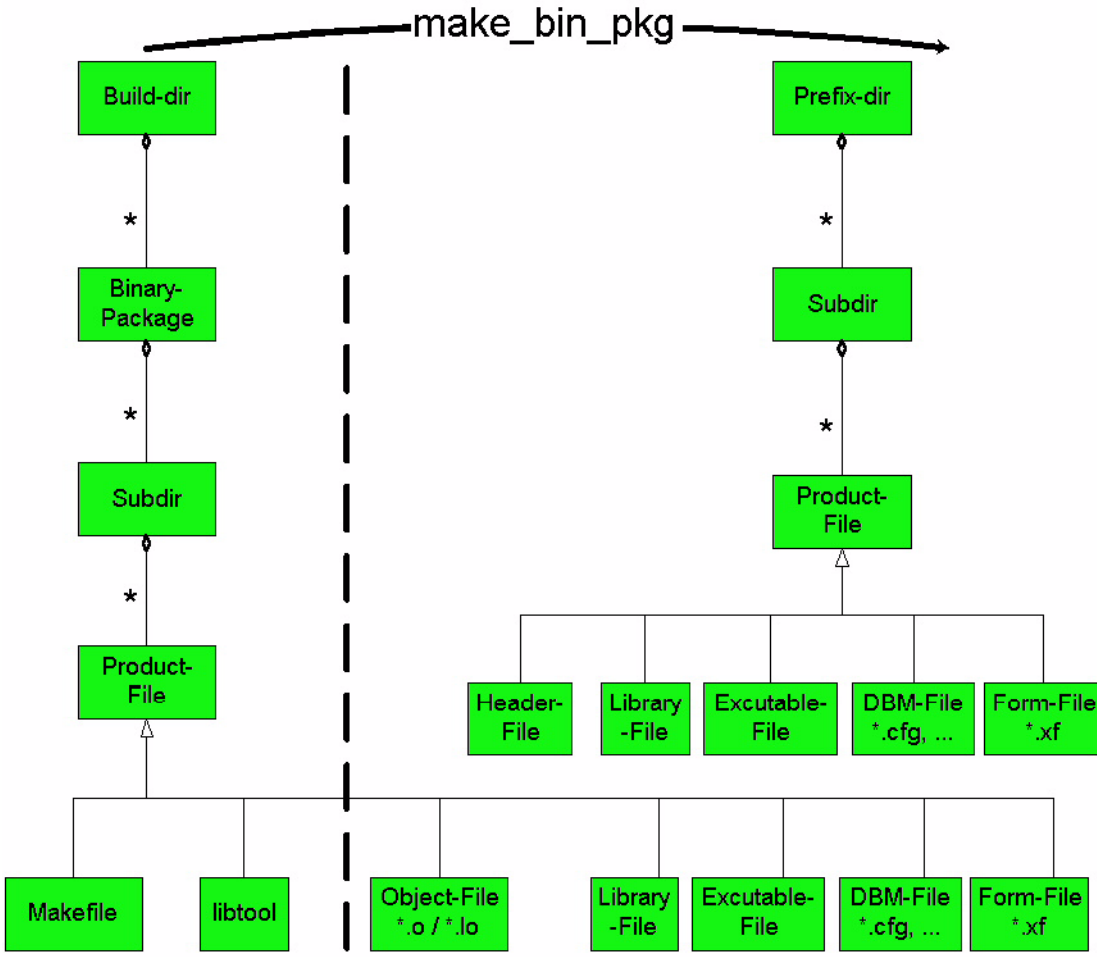


Figure 16:Binary Packages Following Make Install

Index

S

V