OSF Development Environment

User's Guide

ODE Release 2.3.4 **A** (Spring 1995)

Printed on: May 23, 1995

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142

Copyright (c) 1990, 1991, 1992, 1993, 1994, 1995 Open Software Foundation, Inc.

This documentation and the software to which it relates are derived in part from materials supplied by the following:

Copyright (c) 1989, 1990 Carnegie-Mellon University

Permission is hereby granted to use, copy, modify and freely distribute this documentation for any purpose without fee, provided that the above copyright notice appears in all copies and that both the copyright notice and this permission notice appear in supporting documentation. Further, provided that the name of Open Software Foundation, Inc. ("OSF") not be used in advertising or publicity pertaining to distribution of the software without prior written permission from OSF. OSF makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Preface

The *OSF Development User's Guide (DUG)* explains the concepts behind behind the OSF Development Environment (ODE), its methodologies, and the use of the tools for source control, compilation, and release. If you are a new user and have decided to familiarize yourself with the ODE, you should read through the introductions and overviews of each chapter, scanning the examples, before getting into the details. Experienced developers can go directly to the topic or subtopic they are interested in and get the details they need. Listed below are the reference pages related to ODE.

The chapters of this document covers four basic aspects of ODE: sandboxes, compilation environments, source management, and the submission of private work to a public backing tree.

Many of the examples in this document use **suzieq** as the user-ID, **proj2.0b1** as the build name, and **latest** as the default build; **pmax** is used as the target machine.

- Chapter 1 of the DUG introduces the basic concepts and terminology of ODE.
- Chapter 2, "Sandboxes" describe what sandboxes are and how one can use them.
- Chapter 3, "Source Control," covers ODE's source control management system. This includes a description on how ODE manages source control, the source control operations available to the user, how to group files into sets, and the operations that can be performed on sets. The tutorial is concluded in this chapter.
- Chapter 4, "Building Software," details the build process.
- Chapter 5, "Submitting," describes how files get placed from the developer's private sandbox into the next public build. This chapter details the **bsubmit** command for submitting files. The tutorial is concluded in this chapter.

Audience

This document is written for developers and release engineers using the ODE tools..

Applicability

This is Version 3.0 of this document. It is accurate to the changes made in ODE 2.3.

Purpose

The purpose of this document is to provide a guide to the new and experienced developer using the OSF development process and tools. It is also intended to provide an introduction to the concepts behind those processes and tools.

Typographic and Keying Conventions

This document uses the following typographic conventions:

Bold — **Bold** words or characters represent system elements that you must use literally, such as commands, flags, and pathnames. **Bold** words also indicate the first use of a term included in the glossary.

Italic — *Italic* words or characters represent variable values that you must supply.

`Constant width` — Examples and information that the system displays appear in the `constant width` typeface.

[ ] — Brackets enclose optional items in format and syntax descriptions.

{ } — Braces enclose a list from which you must choose an item in format and syntax descriptions.

| — A vertical bar separates items in a list of choices.

< > — Angle brackets enclose the name of a key on the keyboard.

... — Horizontal ellipsis points indicate that you can repeat the preceding item one or more times. Vertical ellipsis points indicate that you can repeat the preceding item one or more times.

This document uses the following keying conventions:

<Ctrl-*x*> or ˆ*x* — The notation **<Ctrl-*x*>** or ˆ*x*, followed by the name of a key, indicates a control character sequence. For example, **<Ctrl-c>** means that you hold down the control key while pressing **<c>**.

Reference Pages

The following reference pages relate to ODE:

bci(1)
bco(1)
bcreate(1)
bcs(1)
bdiff(1)
blog(1)
bmerge(1)
bstat(1)
bsubmit(1)
build(1)
currentsb(1)
genpath(1)
makefiles(5)
make(1)
mklinks(1)
mksb(1)
resb(1)
sadmin(1)
oderc(5)
sbinfo(1)
uptodate(1)

workon(1)

These can be found in appendix A of this document. Reference pages also exist in Section 3 for the library routines found in **libsb.a**.

Problem Reporting

If you have any problems with the software or documentation, please contact a member of the Release Engineering and Distribution group. For ODE users outside OSF, you can send mail to the Release Engineering group at ode-info@osf.org.

1. Introduction

The OSF Development Environment User's Guide (DUG) is part of a collection of documents that describe the OSF Development Environment (ODE). Other documents include the *OSF Development Environment System Administration Guide* and the man pages.

1.1 The OSF Development Environment

The OSF Development Environment (ODE) is designed to allow simultaneous development on multiple revisions of a single set of sources to be compiled for a variety of different and essentially incompatible hardware platforms. At the same time it must satisfy the needs to ship complete, reproducible systems to customers while being flexible enough for individual development.

Three basic areas to ODE are:

- Source code control
- Build Environment
- Private and Public work areas

Each area must allow the individual developer to work independently of other developers, yet allow Release Engineering to bring this work together on a regular basis to create systems for testing and release.

The source control mechanism protects a public revision of each file while allowing developers to simultaneously modify private revisions. It provides controls so these private changes, once they have been built and tested, can be merged back together and integrated with the public revision. Finally, source control provides a method for reproducing earlier releases for bug fixes.

The build environment works with source control to facilitate the building of systems for both release and individual development. It provides a standard set of tools, header files, and libraries allowing official systems to be duplicated while giving developers the flexibility to insert their own tools and files as needed. Much of the details of how a system is to be built is embedded in the ODE common makefiles and the build tools. Within ODE the build environment "knows" a great deal about how the system is to be built.

The ODE build processes are based on the UNIX[1] command **make** and associated **Makefiles**. ODE uses an enhanced version of **make**, which has functionality to support the build environment.

1. UNIX is a registered trademark of UNIX Systems Laboratories, Inc. in the U.S. and other countries.

1.1.1 Terminology

You should understand the following terms before proceeding:

build A front-end to **make**.

a *A complete set of sources, objects, and binaries which have been "built" and put in one place.*

default build For each project there is a default build that contains the newest revisions of submitted sources. In this document, that build is referred to as *latest*; see the supplemental DUGs for the actual name of the default build for each project.

defunct This refers to making a file that is under source control inactive. Once it is made defunct, it can no longer be checked out for editing.

outdating Removes from source control all references to work done on a file in a private sandbox.

project A set of sources that have been grouped together and are treated as a unit. Projects can be as large as the OS and as small or smaller than ODE itself. Each project has separate builds and a separate default build from all other projects.

sandbox A private working area that a developer owns and controls.

set A group of sources within a sandbox that can be manipulated as a unit using the ODE tools.

backing trees A complete set of sources, compilation tools, header files, and libraries that support a sandbox.

2. Sandboxes

This chapter describes how to begin working with ODE. It describes the basic development environment supported by ODE, known as a sandbox. After reading this chapter, you should understand what a sandbox is, its relationship to backing builds and other sandboxes, how to retarget a sandbox to different backing builds, and operations typically performed within a sandbox.

2.1 What is a sandbox?

Sandboxes provide developers with an isolated source development environment. Changes made in one developer's sandbox are not visible to other developers working in their own sandboxes. This makes it possible for many developers to simultaneously develop and test code using the same files without interfering with one another. Once a developer is satisfied with their changes he can integrate them into a backing build. When the changes are integrated, they are then publicly available and other developers can develop software based on these changes.

A sandbox initially contains no source files. Typically, a user only populates his sandbox with files they want to change. These files are taken from the backing builds. During building, the sources from the backing build are used except for the files that exist in the user's sandbox. This allows a developer to make changes to a source tree without having to have the entire tree in his sandbox.

A complete development environment requires a backing build. All sandboxes are backed by backing builds. Each backing build provides a full source tree and the tools, libraries, and header files required to build those sources. Together, these comprise a specific instance of the product built from the sources.

There are two general categories of builds: "static" builds and "dynamic" builds. "Dynamic" builds have changeable source trees and usually contain the most recent sources. When changes are made to this kind of build, they will be immediately visible in all sandboxes backed by it. Builds with source trees that do not change are known as "static" builds. Static builds usually represent "known to be stable" reference points in the development of a product.

The combination of a sandbox and a backing build enable a developer to develop and test a small set of sources against a variety of complete development environments. Although a sandbox is originally backed by a particular build, it can freely float from one backing build to another. The user can change which instance of the software product they would like to develop and test against as often as they need to.

When the changes in a developer's sandbox become stable they can be made available to a wider audience to be used as a basis for further development by others. The integration of a developer's changes into a default build is called "submission."

Submitting to a backing build involves merging your changes with the changes other developers may have submitted and resolving conflicts. If there were only one sandbox that submitted changes to a backing chain then there would never be any conflicts in submissions.

If two developers (each in their own sandbox) are working on the same file, a conflict will arise. This conflict is resolved when each user submits source changes to the default build. All submissions are to a public branch in the default build and submissions are done one at a time. Therefore, each user will integrate their changes with the public branch.

Although you can only submit changes to a dynamic build, most static builds have a default build associated with them. Default builds are always dynamic. When submitting to a static build which has a default build, the changes will actually be submitted to the default build.

2.2 Components of a sandbox

There are three basic components of an ODE sandbox: a source tree for sources being edited, an area for building object modules, and an area to hold header files and libraries for commands. Sources are placed in the **src** directory and objects are placed in the **obj** directory. It is not necessary for the user to specify different directories, they are separated automatically by the tools. All operations on the **obj** tree by the tools are done from the source tree.

For most projects, before any libraries can be built, the header files must be collected into an area where the compiler will find them. Likewise, before the commands can be built, the libraries must be built and copied into an area known to the linker. It is not always convenient to refer to these in the **src** or **obj** directory, so these headers and libraries are copied into an **export** directory. The backing build will have a fully populated export area containing all headers and libraries necessary to build the product and, in most cases, it is not necessary to populate a sandbox **export** directory. Any headers or libraries being developed in a sandbox, however, must be in the **export** directory if you are building other commands with them. The procedure for moving header files and libraries into the **export** directory is covered in the chapter on the build environment.

The layout of a sandbox is shown in Figure 1. The elements within solid boxes are related to source control, and the elements within dashed boxes are related to the ODE build environment.

Sandbox Directory Structure

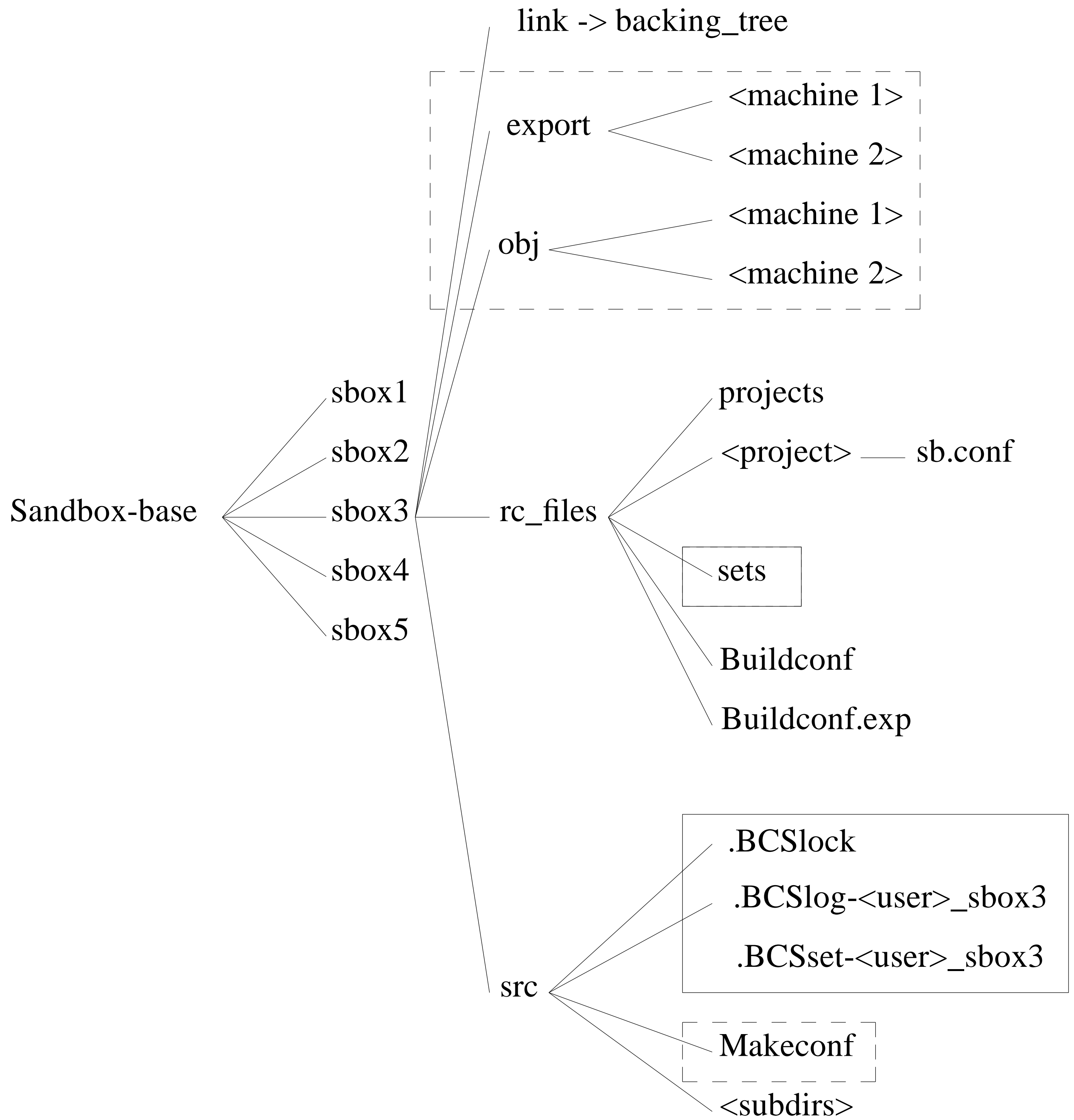


Figure 1.

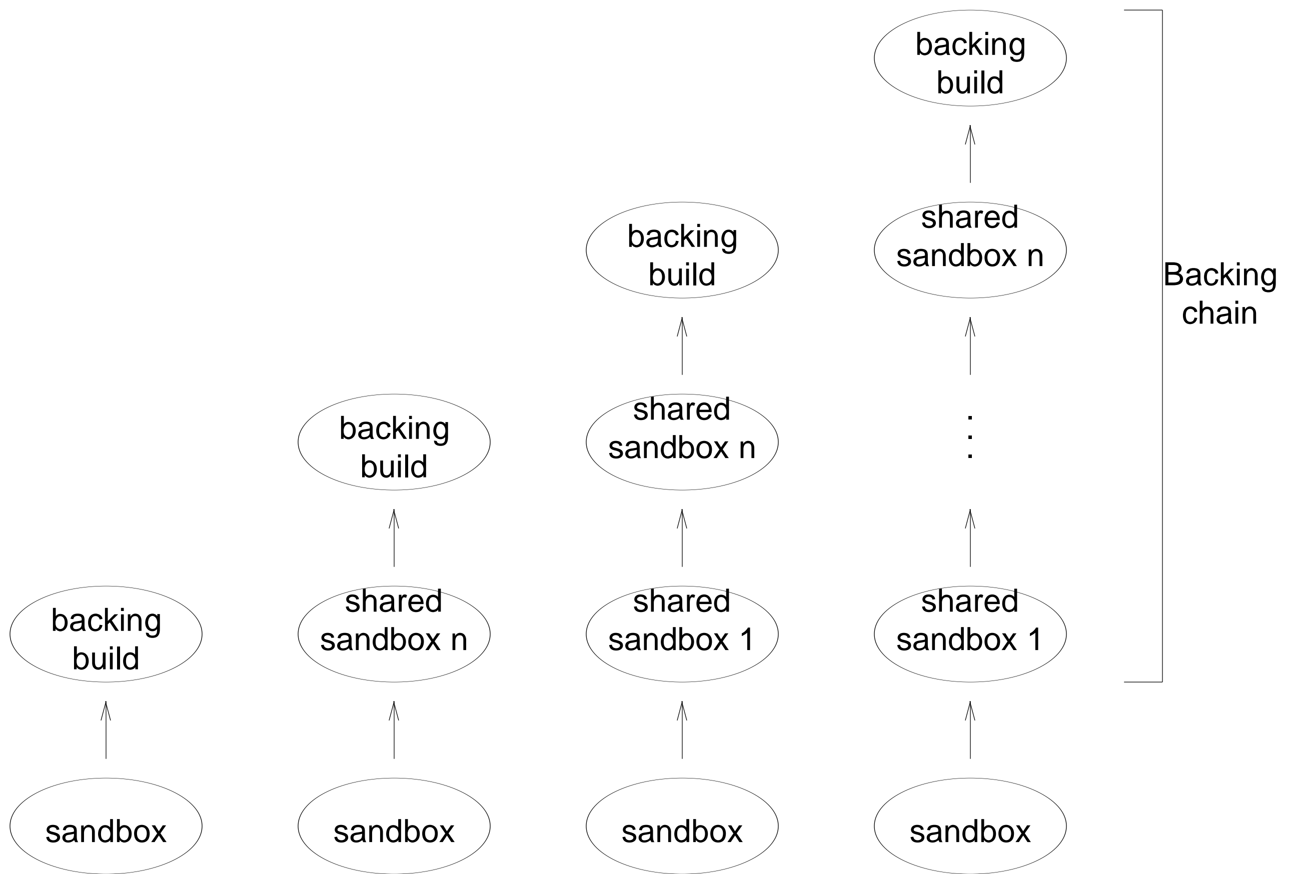
2.3 Chaining sandboxes and backing builds

Sandboxes can be chained together. That is, a sandbox can be set up that is backed by another sandbox, which may be backed by yet another sandbox. These sandboxes can be chained together to any depth as long as the last link in the chain is a full backing build.

Chaining sandboxes is beneficial when it is important to stage integration and visibility of source changes into different levels of development. Each level in the chain can represent an integration point for a different group of users. Also, each level up the chain can represent some subset of a user community, the lowest level representing the smallest set, that is, a developer and his private sandbox. The highest level representing the largest audience, that is, the backing build available to everyone.

The actual structure of the chain and the function of each level in the chain is up to the group of people maintaining it. The essence of each level in a sandbox/backing build chain is to provide an area of private development for a small group of people and allow submission of those changes to a wider audience.

The structure of a backing chain is shown in Figure 2.



A backing chain is any number (including zero) of shared sandboxes with a backing build at the end.

Figure 2.

2.4 The .sandboxrc File

Each ODE user has a file that contains information about the sandboxes the user accesses. Such information includes: the list of the user's sandboxes, the base directory to each sandbox, and the user's default sandbox. It can also contain default arguments for any of the ODE commands. This file is usually located in **${HOME}/.sandboxrc**.

The entries in the **.sandboxrc** file for ODE commands have the following format:

cmdname option ...

It is possible to place the **.sandboxrc** file in a directory other than **${HOME}**. However, each time any of the ODE tools that access this file are used, the user has to specify the new path.

2.5 Operations within sandboxes

The operations on sandboxes supported by ODE include: creating sandboxes, working in sandboxes, populating sandboxes, retargeting sandboxes to different backing trees, and removing sandboxes. Here we will discuss these operations within ODE and the tools available to perform them.

2.5.1 Creating Sandboxes

The command to create a sandbox is **mksb**. This command is the first step in setting up a development environment under ODE. **mksb** creates the sandbox structure which includes the directories **src**, **obj**, **export**, and a directory **rc_files** that maintains datafiles that refer to a backing build or shared sandbox. **mksb** also creates the symbolic link **link** that points to the backing build.

Sandboxes can be created to support any or all of the different machine types. **mksb** creates each of these directories for each machine type listed with the command line option **-m**.

You should be aware of the following restrictions when creating a sandbox:

- Sandbox names cannot contain dashes '-', periods '.', or slashes '/'.
- Each execution of **mksb** creates only one sandbox. Do not specify more than one sandbox name on the command line.
- Each sandbox must be backed by an existing backing build or sandbox. You cannot create a null sandbox and retarget it later.
- Since sandbox names are listed in the file **${HOME}/.sandboxrc** file and the **.sandboxrc** file maps sandbox names to the directories where they reside, a user cannot have more than one sandbox with the same name.
- Moving a sandbox to a different directory will require the following changes: symbolic links created to populate the sandbox must be recreated, and the **.sandboxrc** file must to be updated to show the new sandbox base.

Examples:

The following command is used to create the sandbox **symphony** in the current working directory backed by the osc1.1 backing build for the machine type pmax.

mksb -back /project/osc/build/osc1.1 -m pmax symphony

The following command is used to create the same sandbox but, supports building for the machine types **pmax, mmax** and **at386**.

mksb -back /project/osc/build/osc1.1 -m pmax:mmax:at386 symphony

2.5.2 Working on a Sandbox

workon is used to get into a sandbox environment for editing, compiling, and linking. **workon** sets the user up in a new shell, establishes the proper environment for working in the sandbox, and places the user in the sandbox **src** tree.

The command syntax is simple,

workon [-sb *sandbox-name*]

2.5.3 Populating a Sandbox

Sandboxes, by design, do not require sources in them to build. The ODE tools go through a search path to find all sources required to build. The sandbox is searched, and if sources are not available, each successive link in the backing chain is searched. The last link in the backing chain is a backing build with a fully populated source tree. We recommend that users of ODE do not populate sandboxes with sources, rather, the only sources found in a sandbox should be those checked out for editing.

There are times when populating a sandbox may be desirable to the user (such as source browsing). The ODE tools do support populating of sandboxes in two ways. Setting up symbolic links to sources in the backing chain and physically copying sources from the backing chain. Given a choice between the two, we recommend using symbolic links rather than physical copying of files.

When you create a sandbox, you can specify to **mksb** to put links to the backing tree in the sandbox for all sources, exported files, and tools. You can also have these files copied into the sandbox. Copying them into place limits your ability to retarget your sandbox to other backing trees.

Whether you copy files from the backing tree or merely create links from the sandbox, you should be aware of the advantages and disadvantages of each. Copying files from the backing build to a sandbox takes up a large amount of space and, if it is a dynamic backing build, the sources copied can easily become out of date. Symbolic links take up less space, however, they can pose problems if the sandbox is ever moved around on the file system. Each sandbox contains a link to the backing build appropriately named **link**. It is through this link that all tools refer to the backing build. Each link in a populated sandbox refers to the backing tree through the path *absolute-pathname-to-sandbox*/**link/...**. If the sandbox is moved then the absolute pathnames to the sandbox in each symbolic link becomes invalid. Each of these links will need to be recreated.

The **mklinks** command can also be used to populate a sandbox. This tool provides more flexibility in that you can use it to link individual sources, objects, tools, and headers. You can populate an area with a single file or and entire subsystem By default **mklinks** uses symbolic links for each file it creates.

Example:

The commands:

cd usr/bin
mklinks date

will create symbolic links to all of the sources for the **date** command.

2.5.4 Retargeting a Sandbox

Retargeting a sandbox refers to changing the shared sandbox or backing build that a developers sandbox is backed by. As mentioned earlier, sandboxes rely on backing chains for a complete development environment. Since each backing chain represents a different point in the development of a software product, developers may find it useful to *switch* to a newer backing chain as time goes on.

When you want to be backed by a different build or sandbox, you need to retarget the sandbox using **resb**.

Example:

resb -sb symphony /project/osc/build/osc1.2

2.5.5 Removing a Sandbox

If a mistake has been made in creating a sandbox, the user can **-undo** with **mksb**.

Example:

mksb -undo symphony

2.6 Accessing the Builds

You should consult the release engineers on your project to locate the builds are and how to access them. Usually, all the builds for a single project will be available under a single directory; however, as this is a project-by-project decision, this may not necessarily be the case.

2.7 Split Sandboxes

A user's sandbox can be set up to be backed by multiple backing builds on a directory by directory basis. This is accomplished with the 'projects' file which resides in the rc_files directory. The contents of the projects file is a list of directories and projects. The first field is the directory and the second field is the project to use for that directory.

Along with the projects file, you will need an sb.conf file for every project listed in the projects file. In sandboxes created with ODE 2.3 mksb, there will already be a single line

in the projects file corresponding to the project of the backing build which the sandbox is backed by. There will also be an sb.conf file for that project. You will only need to modify the projects file and add an sb.conf file if you wish to have a sandbox backed by more than one project.

The sb.conf file has 4 entries:

backing_project The name of the project that the sandbox is backed by.

backing_build The logical path used to access the top of the backing build.

ode_sc Indicates whether ODE source control is being used, set to ’true’ or ’false’.

ode_build_env Indicates whether the ODE build environment is being used, set to ’true’ or ’false’.

3. Source Control

This chapter gives an overview of how source control works in ODE. This chapter is divided into three sections. The first section gives an overview of ODE source control. The second section discusses the tools and operations that apply to source files. Such operations include creating new source, checking out files, deleting obsolete source. The last section describes the grouping of source files into sets and operations performed on sets.

3.1 The ODE Source Control Structure

Source control in ODE involves maintaining a revision history of the source being developed as well as integrating source developed in sandboxes to backing builds. The history of a file is maintained to isolate and retrieve specific instances of the software being developed. During the life cycle of a product, it is not uncommon to have multiple development groups working on a common set of sources. For instance, at OSF, groups of developers may be working on many different instances of OSF/1. OSF/1 is currently being developed for release 1.1 while snapshot (bugfix) releases 1.0.1 and 1.0.2 are being developed based on release 1.0. Software quality testing may be done on the latest versions of the sources while benchmarking and test validation may be done on some earlier stable milestone revision. Each of these tasks needs to be performed on a specific instance of the source being developed and requires a revision control system.

ODE uses **RCS** as the basis for its source control. **RCS** is a basic revision control system available on most implementations of Unix.

3.2 Source Control tools and operations

The ODE source control commands support a basic set of operations to maintain revision history. These include: creating files, checking files out for editing, checking files in, deleting files, and getting file status information. These operations are available in most configuration management systems. Here we will discuss these operations within ODE, the tools available to perform them, and some hints on general usage.

3.2.1 Creating Files

Files are created in the source control system through the **bcreate** command. **bcreate** will create the new file under source control, create a path in the current sandbox (if necessary), and put an initial version in your sandbox that is ready for editing.

At this point you can begin to do software development on the file.

The syntax for this command is

bcreate */path/filename*

There are three ways to indicate the location of a file to the ODE tools. By specifying the filename, the tools assume that the file is located in the current directory. By specifying a relative pathname, the tools assume that the file is located in a directory relative to the current directory. By specifying an absolute pathname the tools assume that the file is located in a directory relative to the **src** directory at the base of the sandbox.

The ODE tools expect header information to precede each file. This header is to optionally include copyrights or markers for copyright information and a section describing the history of the file. Details of the copyrights can be found in the project specific DUG as well as the **bci** man page.

If your project is using full copyrights, the headers have the following format:

*comment-leader***COPYRIGHT NOTICE**
*comment-leader***<copyright 1>**
*comment-leader***<copyright 1 continued>**
*comment-leader***<copyright 2>**
*comment-leader***<copyright 3>**
*comment-leader***<copyright ...>**
*comment-leader***HISTORY**
*comment-leader***$Log: $**
*comment-leader***$EndLog$**

If your project is using copyright markers, the headers have the following format:

*comment-leader***@OSF_COPYRIGHT@**

*comment-leader***HISTORY**
*comment-leader***$Log: $**
*comment-leader***$EndLog$**

Since the ODE tools expect to find these markers in each file (and they must be found within a comment), it is important to indicate to **bcreate** what the comment leader will be. For example the headers for a .c file would be:

```
/*
 * @OSF_COPYRIGHT@
 */
/*
 * HISTORY
 * $Log: source.control.tbl,v $
 * $EndLog$
 */
```

and for a Makefile:

```
# @OSF_COPYRIGHT@
#
# HISTORY
# $Log: source.control.tbl,v $
# $EndLog$
```

In most cases **bcreate** will be able to pick the correct leader based on the suffix of the filename, for example, **.c** files are given the leader * (an asterisk). A list of comment leaders and associated filename suffixes is maintained in the rc File **sc.conf**. If **bcreate** cannot find an appropriate filename suffix entry, it will prompt you for the comment leader.

If a file type does not support comments then you should use the comment leader of **NONE**, which tells ODE not to look for headers in the file. The **BIN** leader will tell ODE that the file being created is a binary file and that it should not have a header.

If, for any reason, you need to remove a newly created file from source control, you can use the **-undo** option with **bcreate**. Be warned that this option is intended as a quick way of removing newly created files. Once people have started making changes to a file, **bcreate -undo** will no longer be able to remove the file from source control.

To create a file in the current directory type

> **bcreate** *food.c*

If you find a typo in the filename or it has been placed in the wrong directory, newly created files can be removed by typing

> **bcreate -undo** *food.c*

Create the file with the correct filename

> **bcreate** *foo.c*

3.2.2 Checking Out Files

The **src** tree in a sandbox is initially empty. Sources can be incorporated into a sandbox by retrieving them from the source control system for viewing (read-only, no editing) or for editing (also referred to as "locked"). Retrieving a file from source control it is referred to as **checking out** a file.

Files are checked out with the **bco** command.

To check a file out for editing the following command is used:

> **bco** *myfile.c*

To check out a file for reading only use the following command:

> **bco -read** *anotherfile.c*

3.2.3 Checking In Files

The command for **checking in** files is **bci**. Within a sandbox you can check out a file, edit it, check it in, check out the file again, edit it, and so on as many times as you want and always know the changes to the file remain local to your sandbox.

The check-in operation validates the copyright and history markers and prompts you for log information. The log message can be put on the command line with the **-m** option.

If the file foo.c has been checked out for editing, it can be checked in with the command

bci *foo.c*

If you wish to check in the file and skip the copyright/history checking then first set the comment leader to ’NONE’ as follows:

bcs -c NONE *foo.c*

You can specify the comment on the command line with the command

bci -m "*This is an example comment***"** *foo.c*

3.2.4 Deleting Files

Files can be deleted in one of two ways: **outdate** the private work done to the file in your sandbox, or make the file **defunct** so that it remains under source control, but is no longer accessible and no submissions can be made to it.

Files can be outdated from your sandbox by using the **-o** option to **bcs**. Outdating a file causes the file to be removed from the sandbox and the work that was local to the sandbox for that file is removed from source control. If there is important work done to the file that was not submitted, then outdating the file will cause this work to be lost.

One way to delete a file is simply to make it inactive, or defunct. A defunct file remains under source control (along with its history) but is no longer part of the backing tree.

To defunct a file, use the **-defunct** option with **bci** and submit the file to the default build using **bsubmit**.

Making a file defunct in the default build does not prevent you from successfully checking out an older revision of the file. This is done intentionally so that development can continue for snapshot (bugfix) releases based on earlier revisions of the source.

To make a file defunct, use the following commands:

bci -defunct *foo.c*

bsubmit *foo.c*

3.2.5 Displaying File Status

There are a number of commands that indicate the current status of the files and their branches. The primary commands for getting information are **blog** and **bstat.** These commands print check-in and submission information. The user can find out what revisions exist, the changes checked into a local sandbox, and changes submitted to a backing build.

To print the log header and revision history information for your branch type

blog foo.c

To print the revision that is on the end of the branch labeled OSC1_1 type

bstat -r OSC1_1 -R -V foo.c

3.2.6 Maintaining files under Source Control

The **bcs** command is used for general maintenance of files under source control such as outdating files, labelling revisions, and for changing a comment leaders.

A private development branch can be outdated with the command

bcs -o foo.c

The branch 1.2.3 can be outdated with the command

bcs -o -r 1.2.3 foo.c

Finally, a comment leader can be changed with the following command

bcs -c ' * ' foo.c

See the **bcs** man page for more details.

3.3 Merging revisions

The **bmerge** command provides an automated mechanism for merging two revisions of a file with their common ancestor. The output of the merge is supposed to be the combination of changes made to each file since it diverged from the common ancestor. The tool, however, is not always able to discern the correct combination of changes, especially if the same line has been changed in two different manners. The **bmerge**, command, therefore, sometimes requires the user intervention to resolve these conflicts.

In most cases people are not interested in merging specific revisions of files. Keeping track of the various revisions of a file in which to merge is difficult and error prone. People are much more interested in keeping the file in their sandbox in-sync with submissions made to the backing build. This can be done with the command

bmerge foo.c

See the chapter on submitting for more information on merging.

3.4 Sets

Performing the operations described above is relatively simple and straight forward. When the number of files being checked out, checked in, or submitted is small then using

these tools is simple. With a large system being developed it helps to be able to break the sandbox into groups of manageable working parts. Within ODE, files can be grouped into **sets** whereby each of the operations described above can be applied to the set in one invocation of each command. A user can define a set consisting of all files specific to a subsystem, check them out for editing with one **bco** command, edit, check them in with one **bci** command, and submit them with one invocation of **bsubmit**.

When a sandbox is created a set is automatically defined that includes all the files in the sandbox. This set is the **default** set. Additional sets can be added to the sandbox, but at the very least, there will always be the default set.

Setnames have the same restrictions as sandbox names. Namely, non-shared sandboxes are made up lower case letters. Shared sandbox setnames contain capital letters. Users should not begin a set name with a capital letter unless the set is to be shared.

All files are referenced through a combination of the user-id and setname.

Keep in mind that when private branches are created they are labeled with and referenced through the setname. All operations on this private branch must be referenced through that setname.

Sets are referenced in one of two ways; by entering a set (via **workon**) or by the **-set** option available to most ODE commands. An individual file can be referenced by specifying its name on the command line, or, all files can be referenced by using the **-all** option.

A user can create and enter a set as follows:

> **workon -set** *setname*

A user can also access files in another set as follows:

> **bco -set** *setname* **-all**

To list the sets in the sandbox use the command

> **workon -list**

To remove a set use the command

> **workon -undo** *-set setname*

Files are added to and removed from this set when the file is first check out and when it is submitted to a backing build.

If you have been working on a set of files within a set and wish to delete this work, you can *outdate* each of the private branches that contain this work.

The **-o** option to **bcs** typically deletes, or outdates a private branch associated with a set.

The **-all** option can be used with **-o** to remove the private branches for all files in the set.

4. Build Environment

This chapter covers the principles for building within an ODE sandbox. This includes ODEs use of **make**, the organization of **makefiles**, how **make** uses sandboxes and backing chains, the ability for one to tailor a build environment, and building through **passes**.

ODE uses **make** for building. Backing chains and sandboxes support the building model found on most Unix systems; building a set of source files using a makefile and **make**.

The following are fundamental elements of building in ODE:

- **build** and **make** are used for all building.
- **build** and **make** derive much of their information from environment variables defined in sandboxes and backing chains. Change these variables to customize the build environment.
- Sources and built objects are maintained in separate directories.
- Frequently used build rules are collected into common makefiles.
- Each component makefile includes the common makefiles. When building, definitions of variables within the makefile are used to trigger the execution of rules in the common makefiles.

4.1 build: a Front-end to make

The primary command for building software in ODE is **build**. It serves the same function as the command **make** does in a standard UNIX development environment. In fact, **build** is actually a front-end to **make**. The difference between **build** and **make** is that **build** can operate outside the scope of a sandbox. **make** requires one to **workon** into a sandbox before building.

The options for **build** and **make** are mutually exclusive and any options that **build** does not recognize are passed on to **make**.

To see how **build** works; if you want to create the binary wakeup from wakeup.c in the directory

/sandboxes/suzieq/symphony/src/ode/tutorial/wakeup

Then type

build wakeup

build first searches the **src** tree in the sandbox then the **src** tree in the backing chain for the source. Once the source is found, **build** provides the compiler with a search list for libraries and header files. Libraries and header files are searched first in the **export**

directory of the sandbox then in the **export** directory of the backing chain.

The search list passed to the compiler for header files is similar to:

```
-I../../../myinclude -I/sandboxes/suzieq/symphony/src/myinclude
-I/project/build/proj2.0b1/src/myinclude
```

The search list passed to the compiler for libraries is similar to:

```
-L/sandboxes/suzieq/symphony/export/pmax/usr/shlib
-L/project/build/proj2.0b1/export/pmax/usr/shlib
```

You will not find the built objects in the same directory as the sources. All objects are built in the **obj** directory.

4.1.1 Exporting

Components of a system that are used to build other components are placed in the **export** directory.

In most large software projects, many of the header files and libraries needed to build the system are often part of the project itself. While developing the system; libraries are also being developed. These libraries require development and testing before becoming stable enough to be used by the rest of the system. ODE stages the development and availability of these libraries by using the **export** directory in the sandbox. A library that is under development is not available to the rest of the system until a developer places it into the **export** directory. The processing of placing anything in this directory is called **exporting**.

In addition to libraries, it is convenient to group header files into the **export** directory to cut down on the number of search paths passed to the compiler.

4.1.2 Walkthrough of build Execution

The following is a walkthrough of how **build** would compile the program helloworld. All references to command execution will be from the directory **ode/tutorial/helloworld** in a sandbox which contain the files **Makefile** and **helloworld.c**. The sandbox is backed by a full backing build.

The contents of the files **Makefile** and **helloworld.c** are:

```
PROGRAMS                = helloworld

ILIST                   = helloworld
IDIR                    = /tmp/
```

```
.include <${RULES_MK}>
```

and

```
#include <stdio.h>

main() {

printf ("Hello world\n");

}
```

Running **build** illustrates the way the tools search for source, header files, and libraries.

```
% build
relative path: ./ode/tutorial/helloworld.
mkdir ../../../../obj/pmax/ode/tutorial/helloworld
cd ../../../../obj/pmax/ode/tutorial/helloworld
gcc -B -gline -c  -Dmips -D__mips__ -D_SHARED_LIBRARIES -O     \
-nostdinc -I. -I/sandbox/src/ode/tutorial/helloworld \
-I/backing-tree/src/ode/tutorial/helloworld -I-  \
-I/sandbox/export/pmax/usr/include \
-I/backing-tree/export/pmax/usr/include -pic-lib   \
-L/sandbox/export/pmax/usr/shlib \
-L/backing-tree/export/pmax/usr/shlib \
-L/sandbox/export/pmax/usr/ccs/lib \
-L/backing-tree/export/pmax/usr/ccs/lib     \
 ../../../../../src/ode/tutorial/helloworld/helloworld.c
gcc -B -gline -%ld," -warn_nopic -glue"         \
-L/sandbox/export/pmax/usr/shlib \
-L/backing-tree/export/pmax/usr/shlib \
-L/sandbox/export/pmax/usr/ccs/lib \
-L/backing-tree/export/pmax/usr/ccs/lib -o helloworld.X helloworld.o
mv helloworld.X helloworld
```

Note that relative to the current directory in the sandbox, the objects were built in `../../../../obj/pmax/ode/tutorial/helloworld`.

So, the command that **build** runs for compiling and linking **helloworld** consists of:

- The compiler (**gcc** in our example).
- A series of default compiler options
  `-B -gline -c -Dmips -D__mips__ -D_SHARED_LIBRARIES -O -nostdinc`

- Compiler options to search for include files
  `-I. -I/`*sandbox*`/src/ode/tutorial/helloworld \`
  `-I/`*backing-tree*`/src/ode/tutorial/helloworld`
- Standard include files
  `-I- -I/`*sandbox*`/export/pmax/usr/include \`
  `-I/`*backing-tree*`/export/pmax/usr/include`
- A standard linker option (`-pic-lib`).
- Linker options to search for library files
  `-L/`*sandbox*`/export/pmax/usr/shlib \`
  `-L/`*backing-tree*`/export/pmax/usr/shlib \`
  `-L/`*sandbox*`/export/pmax/usr/ccs/lib \`
  `-L/`*backing-tree*`/export/pmax/usr/ccs/lib`
- The source file to be compiled (relative to the **obj** directory)
  `../../../../../src/ode/tutorial/helloworld/helloworld.c`
  Had the source not been found in the sandbox, each **src** tree in the backing chain would be searched.

4.2 Building with Passes

Most large systems require that its components follow some sort of ordering when being built. Components of the system that are used to build other components of the system must, of course, be built before they can be used.

In most large software projects the best example of this is libraries. If there are components of a software system that require the use of one of its libraries then the library must be built first. By the same token, if the system requires header files to be exported then they must be exported before the system is built.

For software projects, ODE supports ordering builds into the following functions:

- setting up header files in the **export** directory
- building libraries and placing them in the **export** directory
- building software components that use these headers and libraries
- installing the built software for testing
- removing unneeded objects and executables from the **obj** directory

Each step in the build order is called a *pass*. Each pass is applied to the **src** tree in succession. The scope within the **src** tree varies from pass to pass, but each pass is must be completed before the next pass begins.

The passes that are supported in ODE are:

- *export* puts any target which can be exported into the export area
- *comp* does just the compilation step without doing *export*.

- *build* does *export* and *comp*.
- *install* puts files in the directories they will be in on an installed system.
- *clean* removes object (.o) files.
- *rmtarget* removes the target (usually a program).
- *clobber* does a *clean* and a *rmtarget*.
- *lint* runs lint on the sources.
- *tags* creates a tags file.

During each pass the **src** tree is traversed. Each makefile in the **src** tree directs an operation to be performed by the pass being done on the target supplied to the **build** command. The pass indicates the operation to be done while the target indicates the component on which to perform the pass. Many targets may be listed in the makefile with each one applicable to one or more passes.

The syntax for specifying a pass to **build** is:

pass
pass_*target*
pass_all
pass_all_tag

where the terms are defined as:

pass The pass to be done. (**export**, **comp**, **clean**, **install**, etc.) If a pass is not specified, the default will be the *build* pass.

target The target to perform the pass on. Whenever the target is left out, the default target **all** is used.

all A special word which tells **make** to perform the action on all targets listed in the **Makefile**.

tag The target to apply the pass to. The legal tags match the tags a project has setup for **SUBDIRS**.

Therefore, to run **lint** on the program **helloworld**, the command is:

build lint_helloworld.c

all causes the pass specified to be applied to all targets in the makefile.

For example:
build clean_all

removes all objects for all targets listed in each makefile of the source tree.

Example:

If you want to export a file such as **stdio.h** then type

```
% build export_stdio.h
%
```

of course, **stdio.h** must be listed in the **Makefile** as a target for exporting.

If you want to export all header files then type

```
% build export_all
%
```

4.3 ODE Makefiles

One significant difference between the standard UNIX development environment and ODE is the use of common makefiles. Common makefiles hold frequently used build rules in one place so they don't have to be duplicated in each makefile in the **src** tree.

Makefiles for most systems are simple. They include the common makefiles and define makefile variables that trigger execution of common rules.

Including the common makefiles requires the line `.include <${RULES_MK}>` in each makefile. It is important that this line appear after all variable defines in the makefile.

The pass, in conjunction with the variable definitions in the makefile, defines which common rules are triggered. Certain variables are used for certain passes.

The comp and build passes perform compilation and derive much of their information from these makefile variables.

- **PROGRAMS** - list of programs to compile and link.
- **OFILES** - list of objects that comprise a program or library.
- **HFILES** - list of header files on which PROGRAMS and OFILES are dependent.
- **MSGHDIRS** - list of message source files to be processed by **mkcatdefs** for the OSF/1 Message Facility.
- **CATFILES** - list of message catalogs to create for the OSF/1 Message Facility.

The export pass builds and places libraries and header files in an export area for subsequent building. This pass derives much of its information from the following makefile variables.

- **INCLUDES** - list of header files to export.
- **EXPDIR** - the relative directory within the export area.

- **EXPLINKS** - list of links to export.
- **EXPINC_SUBDIRS** - list of subdirectories to traverse during export pass.
- **EXPLIB_TARGETS** - export a library. Must be assigned a value with prefix **export_**. For example **export_libsecurity.a**.

The install pass installs an executable or datafile into a release area and sets access rights. It derives much of its information from the following makefile variables:

- **ILIST** - list of executables or data files to install
- **IDIR** - the directory in which to install them.
- **IMODE** - permissions to set on installed executables.
- **IGROUP** - group to set on installed executables.
- **IOWNER** - owner to set on installed executables.

4.3.1 How Passes are implemented in Makefiles

To specify which directories of the tree to process for each pass, ODE uses the makefile variables **SUBDIRS** and *pass*_**SUBDIRS**.

SUBDIRS is used by passes which visit every directory of the **src** tree. In each source directory, the makefile variable **SUBDIRS** lists all subdirectories to traverse.

Makefile will also have a line setting *pass*_**SUBDIRS** to the set of subdirectories which should be processed for *pass*.

During each pass, if **SUBDIRS** is defined, **make** will process each of the subdirectories listed. This is how a pass is applied to an entire source tree.

Some variables used are:

- **SUBDIRS** The list of sub-directories to to be searched.
- **EXPINC_SUBDIRS** The list of sub-directories which should be searched for include files to export.
- **EXPLIB_SUBDIRS** The list of sub-directories which should be searched for libraries to export.

These would appear in a makefile as follows:

```
SUBDIRS = bin ccs dict doc include lib local sbin
EXPINC_SUBDIRS = include
EXPLIB_SUBDIRS = lib
```

4.4 Modifying the Build Environment

Since the ODE build environment revolves around environment and makefile variables, almost all changes to it are made by changing variables.

Generally, where you change a variable will depend on:

- How long you want it to be affected.
    - If you want the change to last only as long as one run of **build**, set the variable on the **build** command line.
    - If you want the change to last for the duration of this login session, affecting all builds during that time, set the variable in the shell environment.
    - If you want the change to last forever, put it in a **Makefile** or in sandbox **rc_files**.
- What part of the source tree you want to be affected.
    - Changes to components are made in the **Makefile**s and common makefiles.
- Whom you want the change to affect.
    - If you want it to affect just you, set it in the **rc_files** in your own sandbox.
    - If you want it to affect every build that is backed by the backing build, set it in a **Makefile** or in the **rc_files** of the backing chain.

Most of the commonly used variables in makefiles can be specified on a per-target basis by prepending the name of the target and an underscore to the variable. So

foo_CARGS=-DMAKETAB

applies **-DMAKETAB** to **foo** and no other target.

4.4.1 Adding a Program Target

To build a new program in a directory that already builds another program, just edit the **Makefile** to add the new program name to the **PROGRAMS** list.

To build a simple version of the **yes** program from a new source file, **yes.c** in the **tc** directory, edit **Makefile**, and change

```
PROGRAMS                        = tc
```

to

```
PROGRAMS                    = tc yes
```

When you run **build**, this will tell the common makefiles to build both programs.

To build a simple version of the **yes** program in a new directory, create a **Makefile** with the line

```
PROGRAMS                    = yes
```

and the line that include the common makefiles. When you run **build**, this will tell the common makefiles to build the new program.

4.4.2 Location for Building Objects

By default, all objects are built in the **obj** tree. If you wish to build objects in another location, change the **OBJECTDIR** variable in the **rc_files/local** file in your sandbox. You may specify either a relative or absolute pathname. If you specify a relative pathname, it must start from the **src** directory.

A special case is that if you set

OBJECTDIR = ""

the objects will be put in the same directory as the sources.

4.4.3 Compiler Flags

The flags passed to the compiler are taken from a list of variables defined on either the command line, or in a makefile. You can change what gets passed to the compiler by overriding a variable defined in one of these places.

For example, changing a compiler option from the command line:

build "CARGS=-DTARGET"

will add "-DTARGET" on the **cc** command line.

The following variables allow different people to change this list.

CARGSSet by the developer on the **build** command line.

CENV Set by the developer as shell environment variables.

CFLAGS Set in **Makefile** by whoever edits it.

4.4.4 Compiler Include Directories

The **cc** compiler accepts a list of **-I** options, each with a directory name. The list of directories specified by these options is searched to find files in **#include** directives in the .c source file. The following variables allow different people to change this list.

INCARGS Set by the developer on the **build** command line.

INCENV Set by the developer as shell environment variables.

INCFLAGS Set in **Makefile** by whoever edits it.

INCDIRS Set in the **rc_files** by whoever edits them.

Example:

build "INCARGS=-I../../corona/flare -I../../corona/spots"

will cause the directories ../../corona/flare and ../../corona/spots (in the sandbox and the backing chain) to be searched for each **#include** in the source files being built.

4.4.5 Build Tools

Which language tools are used for a project is usually decided on a project basis and set in the common makefiles by Release Engineering. They can usually be overridden on the command line, rc_files, or in the makefile by changing the variables that define them.

Example:

If you need to use a different linker than the rest of the project, you can use

LD = gld

The list of tools include: **CC**, **LD**, **AR**, and **RANLIB**.

4.4.6 Linker Flags

The flags passed to the linker are taken from a list of variables defined on either the

command line, in a makefile, or in a file in the **rc_files** directory. You can change what gets passed to the linker by overriding a variable defined in one of these places.

For example, to add an option to the linker from the command line:

build "LDARGS=-nostdlib"

will place "-nostdlib" on the **ld** command line.

The following variables allow different people to change this list.

LDARGS Set by the developer on the **build** command line.

LDENV Set by the developer as shell environment variables.

LDFLAGS Set in **Makefile** by whoever edits it.

4.4.7 Linker Library Directories

The linker accepts a list of **-L** options, each with a directory name. The list of directories extracted from these options is searched for libraries during linking.

A set of variables has been defined to allow different people to set the directory list in various places:

LIBARGS Set by the developer on the **build** command line.

LIBENV Set by the developer as shell environment variables.

LIBFLAGS Set in **Makefile** by whoever edits it.

LIBDIRS Set in the **rc_files** by whoever edits them.

Example:

build "LIBARGS=-L../../export/pmax/usr/lib/corona/flare -L../../export/pmax/usr/lib/corona/spots" will cause these directories to be searched for each library specified on the **ld** command line.

4.4.8 Optimization and Debugging

A separate environment variable, **OPT_LEVEL**, is provided for defining one of the several debugging and optimization options that can be used with both the compiler and

linker.

Because the **-g** option is used by both the **cc** compiler and the **ld** linker, **OPT_LEVEL** is passed to both.

Examples:

To build with debug on:

OPT_LEVEL = -g

To build with optimization:

OPT_LEVEL = -O

To build with neither debug nor optimization, set:

OPT_LEVEL = ""

To specify optimization for just the compiler use the variable **CC_OPT_LEVEL**. To specify optimization for just the linker use the variable **LD_OPT_LEVEL**.

4.4.9 Installing a New Target in Makefile

During the *install* pass, the common makefiles will install any file listed in the **ILIST** variable into the directory specified by the IDIR variable.

Example:

Assume you have built the simple **yes** program in the same directory as the **tc** program (as described in the example for "Adding a New Target"), and now want to install it.

Edit **Makefile** and change

ILIST = tc

to

ILIST = tc yes

and **build** will thereafter install the **yes** program when it installs the **tc** program.

Example:

Assume that you have built simple **yes** program in its own directory, and now want to install it.

Edit the makefile to add the line

ILIST = yes

Henceforward, **build** will install the **yes** program on the *install* pass. (Be sure that this directory is in the **install_SUBDIRS list of the Makefile in the directory above this one.)**

4.4.10 Location to install Programs

The variable **TOSTAGE** defines the base directory where all executables are to be installed. The pathname for executables after the install pass will be **${TOSTAGE}/${IDIR}/***executable*. **TOSTAGE** can be set as an environment variable or passed to **build** on the command line.

As a side effect, objects cannot be rebuilt if **TOSTAGE** is set. If, during the install pass, this variable is set and you find you need to rebuild a command; you must unset this variable before rebuilding.

4.4.11 Changing Variables in the Common Makefiles

Every **Makefile** in the source tree includes the common makefiles which contain the shared rules. A significant advantage of having common makefiles is the ability to make a change in one common makefile and have it "seen" by every **Makefile** which uses those rules. For example, if a serious defect in the optimizer was found which necessitated turning it off, changing the **OPT_LEVEL** variable in the common makefiles to omit the **-O** option would immediately turn optimization off for all compiles backed by that backing build. While particularly advantageous for Release Engineering, which has to build the entire source tree, this feature is also useful for developers. By checking out one of the common makefiles into their sandbox, a developer can change the build rules for all their work with a single edit.

This is not meant to suggest that the first thing a developer should do is modify all the common makefiles. In fact, they are not usually checked out at all, but are picked up from the backing build. Nor are developers the people who usually make permanent modifications to these files. Such changes are normally made by Release Engineering. But there are times when the developer needs to temporarily modify their entire working environment and the **rc_files** aren’t sufficient for the purpose. That’s when a local copy of a common makefile comes in handy.

The syntax of the common makefiles is mostly that of the standard UNIX **make** command, but most of the lines are comprised of variables so they can work with any type of input.

5. Submitting

During development, all work is performed in sandboxes. The work that one developer does has no effect on the work of another developer. As developers complete their work, they update the default build from their sandboxes. When they have updated the default build, their changes are then visible to all other developers backed by that build.

The process of updating the default build from a sandbox is called submitting. The program that handles the submission process is called **bsubmit**.

This chapter describes the general process of submitting files. For complete information on the submit command and its options, refer to the **bsubmit.1** reference page.

5.1 The Submission Process

The goal of any submission is to update a default build to reflect changes made to files in a developer's sandbox. This includes updating the source control system, the default build backing tree, and the submission logs. Since the work of one developer may overlap the work of another developer, this process needs to be coordinated.

bsubmit coordinates submissions in two ways. It prevents developers from submitting the same files at the same time and it detects situations in which the submission of a file would in effect wipe out an earlier submission. The first situation is handled by putting a hold on submitted files.

All files in the process of being submitted are considered "held." That is, while a file is being held for submission by one developer, no other developer may submit it. If any of the files being submitted are held, **bsubmit** will notify the user and exit.

If none of the files being submitted are held, **bsubmit** will proceed to "validate" the submission. The validation stage consists of checking that each file being submitted is in a valid state in the source control system. This stage is explained in detail in the next section of this chapter. After **bsubmit** has validated the submission it checks to see if any of the files require merging.

Merging is necessary because two or more developers may check out and make different changes to the same version of a file. Without the merging, all changes excepting those made by the last developer's submission would be lost.

Let's say that two developers check out and make changes to version 1.1.2.6 of the file wakeup.c. The first developer to submit his changes will not have to perform any special action to preserve his work. His submission will produce version 1.1.2.7 of the file wakeup.c. If the second developer then submits his changes without performing any special actions, he will produce version 1.1.2.8 of the file wakeup.c which will not contain the changes made by the first developer and incorporated in version 1.1.2.7. This situation is handled by performing a "merge."

A merge is required for a file only when it is not a direct descendant of the most recent version in the source control system.

If the ancestor of a file being submitted is the most recent version in the source control system, that is, the local copy was taken from the default build and no one has submitted

in the time between then and now, no merge is required. Otherwise, the changes incorporated in the file being submitted and changes incorporated in the most recent version in the default build are merged into one new file. After the user resolves any merge conflicts, **bsubmit** checks in the merged files and updates their ancestry information.

At this point, all of the files are ready for submission. There are three steps in the submission process for each file. The first step is the check-out step. The file is checked out of the users private branch and the check-in messages for this branch are condensed into one log message for the public branch. The result is then checked-in to the public branch.

The last step in the submission process is to update the copy of the file in the backing build. The files in the backing build are used for doing builds. Once this step has been completed, the file is considered to be submitted.

After all files have been submitted, **bsubmit** updates the appropriate log files and removes the private branches.

bsubmit keeps track of the work that it has done in two special files, the SNAPSHOT file and the bsubmit.log file. The SNAPSHOT file contains a listing of the most recent versions of all of the files in the default build. The bsubmit.log file contains a history of all of the log information entered for all of the files submitted.

At the end of the submission process, the user is asked whether or not she wants to perform an outdate. Outdating removes all of the submitted files from the user's sandbox and all of the user's versions of the submitted files from the source control system. If this point is reached with no errors, **bsubmit** will inform the user that the submission has succeeded and it will exit.

5.2 Preparing to Submit Files

bsubmit has a number of requirements that must be satisfied by all of the files in a submission before any work may take place. To qualify for submission, a file must be:

- In your sandbox and in the source control system
- In your current set, or in the set specified to **bsubmit**
- Read only. This prevents writeable files, which represent un-saved work, from being submitted.

When listing files to submit, you can use the **-all** option or you can enter the filenames on the command line. If you use the **-all**, all of the files in your current set or the set specified to **bsubmit** with the **-all** option will be submitted. Although **bsubmit** does not accept input from standard input, you can keep your own list of files and input it by entering a line like:

> **bsubmit** *-options* **‘cat** *mylist*‘

5.3 Merging Files

If an actual merge is necessary and the program thinks it successfully merged the files it prints:

```
Merge successful
```

A merge which contains conflicts that **bsubmit** could not resolve produces the message:

`Warning:` *num* `overlaps during merge`

where *num* is the number of conflicts it found. After this warning, it will be necessary to edit the file that **bsubmit** created.

You will be given the following prompt:

```
Abort, ok, edit, merge, [r]co, [r]diff  [edit]
```

At this prompt you have the following options:

Abort - exit bsubmit
ok - accept the file as it is and proceed to the next file
edit - edit the file
merge - perform the merge again
co - check out the latest version of the file
rco - check out the user's version of the file
diff - diff the common ancestor against the latest version of the file
rdiff - diff the common ancestor against the user's version of the file

If there are merge conflicts, you will need to edit the file with the conflicts and decide how to resolve the conflicts. The conflicts will be delineated by less-than and greater-than signs separated by equal signs. The pattern is

<<<<<<
code being submitted
======
code currently in the default build
>>>>>>

You need to search through the file for these patterns, decide which lines are correct and delete the rest. Remember to delete the less-than, greater-than, and equal signs. This file becomes the file that will be checked in to the default build exactly as you have edited it.

If you are absolutely sure the copy of the file you are submitting is correct and want to overwrite the existing file without dealing with the merge, you can enter:

rco

followed by

ok

and the program will use the submitted revision without reference to the existing file in the default build or the common ancestor. **WARNING:**, this procedure removes any changes other developers have made to the file unless they were also in the local revision being submitted.

5.4 Local Cleanup

After the submission is complete and the files have been checked into the default build, checked out again to make them public, and the logs are updated, you have the option of cleaning up your local set.

The advantage to outdating the branch is that the next check out will be based on the revision of the file in the backing tree instead of on the local branch you already have. If the file was merged during the submission, the next time you want to work on the file, you will normally want to use the merged revision of the file. To get this revision, you will have to outdate your local branch and be backed by the default build.

5.5 Submission Failure and Recovery

Whenever any part of a submission fails, **bsubmit** prints out one of two messages. If the submission failed, but no actual work has been done, **bsubmit** will print:

```
- No work has been done for this submission.
- No files have been changed in any way.
- The files in this submission are not held.
- The use of the -resub option is not required and will not be recognized.

Exiting bsubmit.
```

If the submission failed and work has been done, **bsubmit** will print:

```
*** RE-SUBMISSION REQUIRED ***

- Source control information is in an intermediate state.
- Re-submit using -resub time [ -date date ]

Exiting bsubmit.
```

where *time* is the time of submission and *date* is the date of submission.

Resubmissions are based on tracks left by **bsubmit** as it works. Each original submission records the user, time, and date of the submission, as well as the files being submitted, in the the **bsubmit.hold** file. During a resubmission, **bsubmit** uses this file to confirm the contents of the original submission. If the owner, time, or date is incorrect, it will not allow the resubmission to continue.

It also will not allow a resubmission with a different set of files than those listed in **bsubmit.hold**. When you resubmit, **bsubmit** will not accept a list of files because it expects to get the list from **bsubmit.hold**.

The second file **bsubmit** looks for during a resubmission is the **tracking** log which lists the steps completed in the original submission. This filename is in the format *H:MM.USER* where *H* is the hour, *MM* is the minute of the submission and *USER* is the user's name. The file is located in the source control tree and is not accessible by users. On a resubmission, **bsubmit** uses this information to determine which steps it has already completed.

Most of the time, **bsubmit** is able to pick up the submission from where it left off and you do not have to be involved in fixing the logs. There are, however, times when it is

necessary to edit the logs or change an entry. While it is safe for you to edit your local logs, the **SNAPSHOT**, **bsubmit.hold**, and **bsubmit.logs** files must be treated more carefully. The program **sadmin** allows safe access to these files, for copying, editing, locking and unlocking them. If you need to work with these files, you should always use this program.

Appendix A

Manual Pages

This appendix contains the following ODE man pages:

bci(1)
bco(1)
bcreate(1)
bcs(1)
bdiff(1)
blog(1)
bmerge(1)
bstat(1)
bsubmit(1)
build(1)
currentsb(1)
genpath(1)
make(1)
makefiles(5)
mklinks(1)
mksb(1)
oderc(5)
resb(1)
sadmin(1)
sbinfo(1)
sup(1)
uptodate(1)
workon(1)

Index

A

accessing
- builds, 2-10

B

backing build, changing, 2-10

backing_build, 2-11

backing_project, 2-11

bci
- options, **3-15**

bcreate
- files, 3-12

bcs
- options, **3-15**, **3-15**, **3-17**

branches
- private, 3-17

build, 4-19
- accessing, 2-10
- backing, 2-10

C

changing backing build, 2-10

checking in files, 3-14

checking out
- files, 3-14

command
- **bci**, 3-14
- **blog**, 3-15

commands
- **resb**, 2-10

comment leaders
- choosing, 3-13
- NONE, 3-15

configuration files
- projects, 2-10
- sb.conf, 2-10

controlling source, 3-12

copyright markers, 3-13

COPYRIGHT NOTICE, 3-13

copyrights, 3-13
- markers, 3-13

creating
- files, 3-12

D

defuncting a file, 3-15

deleting files, 3-14, 3-15

displaying status, 3-15

E

expanded copyrights, 3-13

F

file headers, 3-13

file
  user **rc**, 2-8

files
  checking in, 3-14
  checking out, 3-14
  creating, 3-12
  defuncting, 3-15
  deleting, 3-14, 3-15
  merging, 3-16
  outdating, 3-15
  removing, 3-15

H

headers, 3-13

I

information, file, 3-15

information, log, 3-15

L

links
  symbolic, 2-10

log information, 3-15

M

make, 4-19

managing source, 3-12

merging files, 3-16

mksb
  command, 2-9

N

names, symbolic, 3-15

NONE, 3-15

O

ODE options
  -all, 3-17
  -set, 3-17

ode_build_env, 2-11

ode_sc, 2-11

options
  -defunct, 3-15

OSF_COPYRIGHT, 3-13

outdating files, 3-15

P

populating sandboxes, 2-9

private branch, 3-17

projects file, 2-10

R

RCS, 3-12

removing files, 3-15

retargeting
    sandbox, 2-10

revision control, 3-12

revision information, 3-15

S

sandbox
    populating, 2-9
    retargeting, 2-10

sandboxes, 1-3
    split, 2-10

sb.conf, 2-10

sets
    managing, 3-16
    name restrictions, 3-17

source control, 3-12

source file maintenance, 3-16

split sandboxes, 2-10

status
    displaying and updating, 3-15

symbolic links, 2-10

symbolic names, 3-15

U

updating status, 3-15

user **rc** file, 2-8

W

workon, 3-17
    options, **3-17**, **3-17**

bci command, 3-14

blog command, 3-15

resb
    command, 2-10

-defunct option:, 3-15

-all, 3-17

-set, 3-17

-undo
    bcreate, 3-14

CONTENTS

Preface ii
Audience ii
Applicability ii
Purpose ii
Typographic and Keying Conventions ii
Reference Pages iii
Problem Reporting iv

1. Introduction 1
1.1 The OSF Development Environment 1

2. Sandboxes 3
2.1 What is a sandbox? 3
2.2 Components of a sandbox 4
2.3 Chaining sandboxes and backing builds 6
2.4 The .sandboxrc File 8
2.5 Operations within sandboxes 8
2.6 Accessing the Builds 10
2.7 Split Sandboxes 10

3. Source Control 12
3.1 The ODE Source Control Structure 12
3.2 Source Control tools and operations 12
3.3 Merging revisions 16
3.4 Sets 16

4. Build Environment 19
4.1 build: a Front-end to make 19
4.2 Building with Passes 22
4.3 ODE Makefiles 24
4.4 Modifying the Build Environment 26

5. Submitting 33
5.1 The Submission Process 33
5.2 Preparing to Submit Files 34
5.3 Merging Files 35
5.4 Local Cleanup 36
5.5 Submission Failure and Recovery 36

Appendix A 38

Index 124