Abuild Users' Manual

For Abuild Version 1.1.6, June 30, 2011

Jay Berkenbilt

Abuild Users' Manual: For Abuild Version 1.1.6, June 30, 2011

Jay Berkenbilt

Table of Contents

List of Figures

List of Tables

Notes For Users of Abuild Version 1.0

This manual is written for abuild version 1.1. If you are a user of abuild version 1.0 and are just looking for a summary of what changed, please see Appendix B, *Major Changes from Version 1.0 to Version 1.1*, page 257. The material there includes a summary of a change along with cross references to relevant sections of documentation.

Please note that, with a small handful of exceptions, abuild version 1.1 is be able to build software that used abuild 1.0 with few if any modifications. The section on changes in version 1.1 (Appendix B, *Major Changes from Version 1.0 to Version 1.1*, page 257) includes a detailed list of things to watch out for during upgrading and when running in 1.0-compatibility mode.

How to Read This Manual

Welcome to the abuild manual! You may always find the latest copy of this manual on abuild's website [http://www.abuild.org]. This manual is designed to get you up and running with abuild quickly: the most essential and common topics are presented first so that you can just start at the beginning and stop reading when you feel that you've seen enough to get going. Then, when you are ready, you can come back for documentation on the full depth of abuild's functionality. If you come across something in the first reading that you don't understand, it's probably safe to skip it and come back when you're more comfortable. As each new concept is presented, it is enhanced with examples. A list of all the examples in the document can be found in Appendix L, *List of Examples*, page 337. If you are just looking for changes from previous versions of abuild, please see Appendix A, *Release Notes*, page 225 and Appendix B, *Major Changes from Version 1.0 to Version 1.1*, page 257.

This manual is divided into four parts. Each part of the document draws on material introduced in the earlier parts. Although earlier parts of the documentation are intended to be understandable without the material from the later parts, they contain forward cross references where appropriate.

In Part I, “Quick Start”, page 1, we cover basic information that should help you come up to speed on using abuild for day-to-day work. It is geared toward people who are working on an existing software baseline that uses abuild. In Part I, you will learn about what abuild is and the types of problems it was designed to solve, be introduced to some basic terminology, and see a few examples of how to perform some simple build operations. This part of the manual is very short and is designed to be readable in one sitting. Casual users of abuild may have no need to read past Part I.

In Part II, “Normal Operation”, page 19, we introduce the most common features of abuild. All the basic features are covered, and a few advanced features are covered. All the information you need for simple projects has been presented by the end of Part II.

In Part III, “Advanced Functionality”, page 78, we introduce advanced topics. By the end of Part III, you will have been exposed to every feature of abuild.

Part IV, “Appendices”, page 223 consists of a small handful of appendices.

For those wishing to go still deeper, the abuild source code is heavily commented, and the software comes with a thorough automated test suite that covers every feature of the software and many error conditions as well.

Acknowledgments

The creation of abuild would not have been possible without the enthusiastic support of my employer, Argon ST [http://www.argonst.com]. Argon not only recognized the important role of a strong build tool in contributing to the overall quality and reliability of its software, but saw the value of releasing it to the open source community in hopes of making an even broader contribution.

There are many people within Argon who helped take abuild to where it is now, but among these, a handful of people deserve special mention:

- Brian Reid, who first introduced me to Groovy, the language that is at the heart of abuild version 1.1's significantly improved Java support, and who kept the momentum going for making abuild's Groovy-based Java framework a reality
- Brian Reid, Joe Schettino, Kathleen Friesen, and Brandon Barlow who met with me many times to help hammer out and test early versions of the Groovy-based Java framework
- Brandon Barlow for tirelessly testing numerous builds with abuild 1.1 during its alpha period.
- Cass Dalton, who has frequently served as a sounding board as I think about new abuild capabilities, and who has played a significant role in helping to ensure that abuild is as stable and widely usable as possible
- Chris Costa, who served as a sounding board and contributed numerous ideas throughout the entire development process of abuild, including conducting a thorough review of the abuild 1.0 documentation
- Andrew Hayden, who spent many hours reviewing and critiquing the entire manual prior to the release of version 1.0 and who contributed many feature ideas designed to ease implementation of an abuild Eclipse plugin
- Joe Davidson, the first abuild evangelist who has been invaluable in getting abuild to become as widely accepted within Argon ST as it is
- Gavin Mulligan, who has consistently taken the time to report any problem, no matter how small, and who probably reported more issues than everyone else combined during abuild's pre-1.0 alpha period
- Bob Tamaru, who in addition to being a mentor and supporter for most of my career, provided considerable assistance to me as I presented the case to Argon ST to allow me to release abuild as an open source project

Notes About Documentation Formatting

This manual is written in docbook. The PDF version of the manual was generated with Apache fop, which as of this writing, is still incomplete. There are a few known issues with the PDF version of the documentation. Hopefully these issues will all be addressed as fop matures.

- There are many bad line breaks. Sometimes words are incorrectly hyphenated, and line breaks also occur between two dashes in command line options and even between the two + characters of “C++”.
- In many of the example listings, there are lines that would be longer than the shaded boxes in the PDF output. We wrap those lines and place a backslash (\) character just before and after the extra line breaks. This is done for both the HTML and the PDF output even though the long lines are only a problem for the PDF output.
- Some paragraphs appear to have extra indentation. This is because the formatting software generates a hard space whenever we have an index term marker in the text.
- There are no bookmarks. It would be good if we could create bookmarks to the chapter headings, but as of this writing, the documented procedure for doing this does not appear to work.

Part I. Quick Start

The material contained in this part is geared toward new and casual users of abuild. Without going into excessive detail, this part gives you a quick tour of abuild's functionality and presents a few examples of routine build operations. By the end of this part, you should be able to use abuild for simple build operations, and you should have begun to get a feel for the basic configuration files.

Chapter 1. Introduction

1.1. Abuild Overview

Abuild is a system designed to build large software projects or related families of software projects that are divided into a potentially large number of components. It is specifically designed for software projects that are continually evolving and that may span multiple languages and platforms. The basic idea behind abuild is simple: when building a single component (module, unit, etc.) of a software package, the developer should be able to focus on that component exclusively. Abuild requires each component developer to declare, by name, the list of other components on which his or her component depends. It is then abuild's responsibility to provide whatever is needed to the build environment to make other required items visible.

You might want to think of abuild as an *object-oriented build system*. When working with abuild, the fundamental unit is the *build item*. A build item is essentially a single collection of code, usually contained within one directory, that is built as a unit. A build item may produce one or more products (libraries, executables, JAR files, etc.) that other build items may want to use. It is the responsibility of each build item to provide information about its products that may be used by other items that depend on it. This information is provided by a build item in its abuild *interface*. In this way, knowledge about how to use a build item is encapsulated within that build item rather than being spread around throughout the other components of a system.

To implement this core functionality, abuild provides its own system for managing build items as well as the dependencies and relationships among them. It also provides various build rules implemented with underlying tools, specifically GNU Make and Apache Ant accessed using the Groovy programming language, to perform the actual build steps. We refer to these underlying tools as *backends*. Although the bulk of the functionality and sophistication of abuild comes from its own core capabilities rather than the build rules, the rules have rich functionality as well. Abuild is intended to *be* your build system. It is not intended, as some other tools are, to wrap around your existing build system. [1]

Support for compilation in multiple programming languages and on multiple platforms, including embedded platforms, is central to abuild's design. Abuild is designed to allow build items to be built on multiple platforms simultaneously. An important way in which abuild achieves this functionality is to do all of its work inside of an *output directory*. When abuild performs the actual build, it always creates an output directory named *abuild-`platform`*. When abuild invokes make, it does so in that directory. By actually invoking the backend in the output directory, abuild avoids the situation of temporary files conflicting with each other on multiple simultaneous builds of a given build item on multiple platforms. For ant-based builds (using either the supported Groovy backend or the deprecated xml-based ant backend), each build is given a private ant ***Project*** object whose *basedir* is set to the output directory. Abuild is designed to never create or remove any output files outside of its output directories. This enables abuild's cleanup operation to simply remove all output directories created by any instance of abuild, and also reduces the likelihood of unintentionally mixing generated products with version-controlled sources.

1.2. Typographic Conventions

The following list shows the font conventions used throughout this document for the names of different kinds of items.

`literal text`
`replaceable text`
build items and ***build item scope names***
Abuild.conf keys, **flags**, and **traits**
Abuild.interface variables, *java properties*, and *make variables*
Abuild.interface keywords
commands and **build targets**

[1] Abuild can, however, interoperate with other build systems as needed, which may be useful while transitioning a software development effort to using abuild.

command line options and **build sets**
environment variables
file names and *make/Groovy rule sets*
`platforms`, `platform types`, and `target types`

1.3. Abuild Version Numbers and Release Policy

This section describes what you can expect in terms of abuild version numbers and non-compatible changes.

1.3.1. Abuild Version Numbers

Each abuild release is assigned a version number. For abuild releases, we use the following version numbering convention:

```
major.minor.prerelease-or-update
```

The *`major`* field of the version number indicates the major version number. It changes whenever a major release is made. A new major release of abuild represents a wholesale change in the way abuild works. Major release are expected to be very infrequent.

The *`minor`* field of the version number indicates the minor version number. It changes whenever a minor release is made. A minor release is an incremental release that may introduce significant new features, fix bugs, or change the way some things work, but it will not fundamentally shift the way abuild works. We impose tight restrictions on the introduction of non-backward-compatible changes in minor releases as discussed below.

The *`prerelease-or-update`* field can indicate either a prerelease version or an update release of a specific minor version. A prerelease is an alpha or beta release or a release candidate that precedes a regular release. An update release may contain bug fixes or new features as long as no non-compatible changes are made to existing functionality. Allowing new non-breaking features to be introduced in an update release makes it possible to add features to abuild incrementally while still guaranteeing as much compatibility as possible. There is no support for a prerelease of an update to a specific minor version (like 1.1.1.b1).

Before a regular major or minor release, there may be a series of alpha releases, beta releases, and release candidates. In those cases, the *`prerelease-or-update`* field of the version number is either "`a`", "`b`", or "`rc`" followed by a number. The prerelease version numbers clearly indicate which regular release the prerelease applies to. For example, version `1.3.a4` would be the fourth alpha release preceding the release of version `1.3.0`.

After any major or minor release, it is possible that a small problem may be corrected in a bug-fix release. In such a release, the *`prerelease-or-update`* field contains a number that indicates which bug-fix release this is. For example, version `1.2.1` would be a bug-fix release to version `1.2.0`.

Historical note: the first release of abuild 1.0 was just version 1.0, not version 1.0.0. The use of "x.y.0" was introduced with version 1.1.0 so that "abuild x.y" could unambiguously refer to all update releases of minor version x.y rather than just the first.

1.3.2. Backward Compatibility Policy

In a new major release of abuild (e.g., version 2.0.0), there is no promise that changes will be backward compatible, nor is there any expectation that configuration files from older abuild releases will work with the new version. When possible, care will be taken to mitigate any inconvenience such as providing upgrade scripts.

In each new minor release of abuild, there may be new features and backward-compatible changes. In minor releases, we adopt a stricter policy regarding non-backward-compatible changes. Specifically, non-backward-compatible

changes may be introduced only if the changed construct generated a deprecation warning in the previous minor release. In other words, if particular construct in version `1.3` is going to be dropped or changed in a non-compatible way, the change can't be made until version `1.5`. In version `1.4`, the new way may work, but use of the deprecated construct must still work and must generate a warning. The old way can be dropped entirely in version `1.5` once users have had a chance to adjust their configuration files. In that way, users who take every minor release upgrade can be guaranteed that they will not experience surprise non-compatible changes, and they will not have to update their configuration files at the same time that they upgrade abuild.

With alpha releases, there is no commitment to avoiding non-compatible changes. In particular, a feature that was introduced into abuild during an alpha testing period may be modified in non-compatible ways or dropped entirely during the course of alpha testing. During beta testing, every effort will be made to avoid non-compatible changes, but they are still allowed. No non-compatible changes will be made from the first release candidate through the next minor release.

Specific exceptions may be made to any of the above rules, but any such exceptions will be clearly stated in the release notes or the documentation. It may happen, for example, that a particular new feature is still in development when a release is made. In that case, the release notes may declare that feature to still be alpha, in which case non-compatible changes can be introduced in the next release.

We'll clarify with some concrete examples. Suppose a new feature is planned for version `1.4` of abuild. It would be okay if the first implementation of that feature appeared in version `1.4.a2` and if the feature were changed in a non-compatible way in `1.4.a6`. However, after version `1.4.0` was released, the next non-compatible change would not be permitted until version `1.5.a1`, and even then, the feature as it worked in version `1.4.0` would still have to work, though a deprecation warning would be issued. The old version `1.4.x` way of doing things could stop working altogether in version `1.6.a1`. It is also okay to add a new feature *within* a minor release. For example, it's okay if `1.0.3` adds some feature that wasn't there in `1.0.2` as long as everything that worked in `1.0.2` works the same way in `1.0.3`. In other words, although everything that worked in `1.0.2` must work in `1.0.3` , there's no expectation that everything that works in `1.0.3` must have worked in `1.0.2`.

1.4. Design Features

This section describes many of the principles upon which abuild was designed. Understanding this material is not critical to being able to use abuild just to do simple compiles, but knowing these things will help you use abuild better and will provide a context for understanding what it does.

Build Integrity
: Abuild puts the integrity of the build over all other concerns. Abuild includes several rigorously enforced integrity checks throughout its implementation in order to prevent many of the most common causes of build integrity problems.

Strict Dependency Management
: Build items must explicitly declare dependencies on other build items. These dependencies are declared by name, not by path. The same mechanism within abuild that is used to declare a dependency is also used to provide visibility to the dependent build item. (A build item reads the interfaces of only those build items on which it directly or indirectly depends.) In this way, it is impossible to *accidentally* become dependent on something by unwittingly using files that it provides. Abuild guarantees that there are no circular dependencies among build items and also provides a fundamental guarantee that all build items in a dependency chain resolve names to paths in a consistent way within the dependency tree.

Directory Structure Neutrality
: Build items refer to each other only by name and never by path. Abuild resolves build item names to paths internally and provides path information at runtime as needed. This makes any specific abuild installation agnostic about directory structure and makes it possible to move things around without changing any build rules. In this way, abuild stays out of the way when it's time to reorganize your project.

Focus on One Item at a Time

When using abuild, you are generally able to focus on building just the item you are working on without having to worry about the details of the items it depends on. Abuild does all the work of figuring out what your environment has to look like to give you access to your dependencies. It can then start a local build from anywhere and pass the right information to that local build. This is achieved through encapsulation of knowledge about a build item's products inside the build item itself and making that knowledge available to its users through an abuild-specific interface.

Environment Independence

Abuild does not require you to have any project-specific or source tree-specific environment variables set, be using any particular shell or operating system, or have the abuild software itself installed in any particular location. Abuild is designed so that having the **abuild** command in your path is sufficient for doing a build. This keeps abuild independent from any specific source tree or project. Abuild can be used to build a single-source-file, stand-alone program or an elaborate product line consisting of hundreds or thousands of components. It can be also used for multiple projects on the same system at the same time. No special path settings or environment variable settings are required to use abuild, other than ensuring that the external tools that your build requires (GNU Make, compilers, etc.) are available and in your path.

Support for Parallel and Distributed Builds

When building multiple items, abuild creates a *build set* consisting of all the items to be built. It computes the directories in which it needs to build and invokes the build iteratively in those directories. Abuild automatically figures out what can be built in parallel and what the build order should be by inspecting the dependency graph. Abuild avoids many of the pitfalls that get in the way of parallel and distributed operation including recursive execution, shell-based loops for iteration, file system-based traversal, and writing files to the source directory.

Support for Multiple Platforms

Abuild was designed to work on multiple platforms. It includes a structure for referring to platforms and for encapsulating platform-specific knowledge. This makes it easier to create portable build structures for portable code.

Efficiency

Abuild aims to be as efficient as possible without compromising build integrity. Abuild calculates as much as possible up front when it is first invoked, and it passes that information to backend build programs through automatically-generated files created inside its own output directories. By computing the information one time, abuild significantly reduces the degree to which its backend build programs' rules have to use external helper applications to compute information they need. Abuild's configuration files and build tree traversal steps are designed in such a way that abuild never has to perform unbounded searches of a build tree. This enables startup to be fast even on build trees containing thousands of build items.

Encapsulation

Build items encapsulate knowledge about what is required by their users in order to make use of them at build time. The user may also create build items with restricted scope, thus allowing private things to be kept private. This makes it possible to refactor or reorganize individual components of a system without affecting the build files of other build items that depend on them.

Declarative Build Files

The majority of build item configuration files are declarative: they contain descriptions of what needs to be done, rather than information about how to do it. Most end user configuration files contain nothing but variable settings or key/value pairs and are independent of the platform or compiler used to build the item. For those cases in which a declarative system is insufficient to express what needs to do be done, abuild provides several mechanisms for specific steps to be defined and made available to the items that need them.

Support for Multiple Backends

The parts of abuild that manage dependencies and build integrity are distinct from the parts of abuild that actually perform builds. Abuild current uses either GNU Make or Apache Ant, accessed through a Groovy language front

end, to perform builds. [2] The internal integration between abuild and its backend build programs is fairly loose, and adding additional backends requires relatively minor and localized code changes. In addition, abuild requires only the backends that a particular build tree uses to be present on your system when you are performing a build. That is, if you are building only Java code, you don't need GNU Make, and if you're building only C and C++ code, you don't need a Java or ant environment.

[2] There is also support for ant using xml files. This was the primary mechanism for using ant in abuild 1.0, but it is deprecated in version 1.1 in favor of the much more flexible and capable Groovy-based backend. Throughout this document, we refer to it as the “deprecated xml-based ant” framework.

Chapter 2. Building and Installing Abuild

2.1. System Requirements

You may always find the latest version of abuild by following the links on abuild's website [http://www.abuild.org]. To use abuild, the following items must be available on your system:

- GNU Make [http://www.gnu.org/software/make/] version 3.81 or higher is required if you are building any build items that use GNU Make as a backend. This would include platform-independent code and C/C++ code, but not Java code.
- A Java 5 or newer Java SDK is required if you are going to use abuild to build Java code. Abuild is known to work with OpenJDK 1.6.
- Apache Ant [http://ant.apache.org/] version 1.7.0 or newer is required if you are building any Java code. If you are using abuild's deprecated xml-based ant framework, then you also need ant-contrib [http://ant-contrib.sourceforge.net/] version 1.0.b3 or later installed in either ant's or abuild's lib directory.
- Perl [http://www.perl.com/] version 5.8 or newer is required if you are performing any GNU Make-based builds.
- Perl version 5.8 or newer and qtest [http://qtest.qbilt.org/] version 1.0 or newer are required if you are using the qtest automated test framework. Abuild's own test suite uses qtest. Note also that qtest requires GNU diffutils [http://www.gnu.org/software/diffutils]. Any version should do.
- In order to use abuild's autoconf support, you need autoconf [http://www.gnu.org/software/autoconf] version 2.59 or newer, automake [http://www.gnu.org/software/automake] version 1.9 or newer. These are also required for abuild's test suite to pass since the test suite exercises its autoconf support.
- If you are planning on building any GNU Make-based build items on Windows, Cygwin [http://www.cygwin.com/] is required. For a Java-only abuild installation on Windows, Cygwin and Perl are not required. It is hoped that a future version of abuild will not require Cygwin. For details on using Cygwin with abuild, please see Section 2.4, “Additional Requirements for Windows Environments”, page 8.

To build abuild, you must also have version 1.35 or newer of boost [http://www.boost.org/]. Abuild uses several boost libraries, including regex, thread, system, filesystem, and date_time as well as several header-only libraries such as asio, bind, and function. Abuild is known to buildable by gcc and Microsoft Visual C++ (7.1 or newer), but it should be buildable by any compiler that supports boost 1.35. In order for shared library support to work properly with gcc, gcc must be configured to use the GNU linker. [1] Abuild itself contains C++ code and Java code, so all the runtime requirements for both systems are required to build abuild.

In order to build abuild's Java code, which is required if you are doing any Java-based builds, you must have at least version 1.5.7 of Groovy [http://groovy.codehaus.org]. It is recommended that you have at least version 1.6.0. It is not required that you have Groovy to *run* abuild because abuild includes an embedded version of the Groovy environment, but a full installation of Groovy is required in order to do the initial bootstrapping build of abuild's Java code. [2]

As of abuild version 1.1.0, abuild is known to work with Groovy versions 1.6.7 and 1.7-RC-1, which were the latest available versions at the time of the release. Upgrading abuild's embedded version of Groovy is as simple as just replacing the embeddable Groovy JAR file inside of abuild's lib directory. Just delete the old one and copy the new

[1] The only reason for the GNU linker requirement is that abuild currently knows about **-fPIC**. It would be better to have a more robust way of configuring flags for position-independent-code, but it's not clear how to do this without replicating all the knowledge built into libtool or having some autoconf-like method of configuring abuild at runtime.
[2] Besides, every Java programmer should have a copy of Groovy installed!

one in. abuild will automatically find it even though its name will have changed to include the later version number. Ideally, you should also rebuild abuild's java support from source and rerun abuild's test suite just to be sure abuild still works properly with the latest Groovy.

Since abuild determines where it is being run from when it is invoked, a binary distribution of abuild is not tied to a particular installation path. It finds the root of its installation directory by walking up the path from the abuild executable until it finds a directory that contains *make/abuild.mk*. This makes it easy to have multiple versions of abuild installed simultaneously, and it also makes it easy to create relocatable binary distributions of abuild.

Abuild itself does not require any environment variables to be set, but ant and/or the Java development environment may. If you have the *JAVA_HOME* and *ANT_HOME* environment variables set, abuild will honor them when selecting which copy of java to run and where to find the ant JAR files. Otherwise, it will run java and ant from your path to make those determinations. Although abuild is explicitly tested to work without either *ANT_HOME* or *JAVA_HOME* set, if any Java builds are being done, abuild will start up a little more quickly if they are set. As many other applications expect these to be set, it is recommended that you set *JAVA_HOME* and *ANT_HOME*. When abuild invokes Java for any of the Java-based backends, it will automatically add all the JAR files in *$ANT_HOME/lib* to the classpath as well as all JAR files in abuild's own *lib* directory. Abuild includes a copy of Groovy's embeddable JAR in its own *lib* directory. You can copy additional JAR files into *lib* as well, but if you do so, just remember that those JAR files will not automatically be available to users whose abuild installations do not include them.

As you begin using abuild, you may find yourself generating a collection of useful utility build items for things like specific third-party libraries, external compilers, documentation generators, or test frameworks. There is a small collection of contributed build items in the *abuild-contrib* package, which is available at abuild's web site [http://www.abuild.org]. These may have additional requirements. For details, please see the information about *abuild-contrib* on the website.

2.2. Building Abuild

Abuild is self-hosting: it can be built with itself, or for bootstrapping, it can be built with a GNU Makefile that uses abuild's internal GNU Make support. To build abuild's Java code, you also need Groovy, Apache Ant and a Java development environment. Please see the file *src/README.build* in the source distribution for instructions on building abuild.

2.3. Installing Abuild

If you are creating a binary distribution or installing from source, please see the file *src/README.build* in the source directory. If you are installing from a pre-built binary distribution, simply extract the binary distribution in any directory. Abuild imposes no requirements on where the directory should be or what it should be called as long as its contents remain in the correct relative locations. You may make a symbolic link to the actual *bin/abuild* executable from a directory in your path. Abuild will follow this link when attempting to discover the path of its installation directory. You may also add the abuild distribution's *bin* directory to your path, or invoke abuild by the full path to its executable. [3]

2.4. Additional Requirements for Windows Environments

To build abuild and use it in a Windows environment for make-based builds, certain pieces of the Cygwin [http://www.cygwin.com/] environment are required. [4] Note that abuild is able to build with and be built by Visual C++ on

[3] If abuild is not invoked as an absolute path, it will iterate through the directories in your *PATH* trying to find itself. Therefore, abuild may fail to work properly if you invoke it programmatically, pass \"`abuild`" to it as `argv[0]`, and do not have the copy of abuild you are invoking in your path before any other copy of abuild. This limitation should never impact users who are invoking abuild normally from the command line or through a shell or other program that searches the path.

[4] This may cease to be true in a future version of abuild.

Windows. It uses Cygwin only for its development tools. Cygwin is not required to run executables built by abuild in a Windows environment, including abuild itself. However, Cygwin is required to supply **make** and **perl** to abuild. The following parts of Cygwin are required:

Devel

autoconf
automake
make

System

rebase

Util

diffutils

Perl is required, but appears to be installed by default in recent Cygwin installations.

Note that rebaseall (from the rebase package) may need to be run in order for *fork* to work from perl with certain modules. (Although abuild itself doesn't call *fork* from perl, qtest, which is used for abuild's test suite, does.)

Other modules may also be desirable. In particular, *libxml2* from the *Text* section is required in order to run certain parts of abuild's test suite, though the test suite will just issue a warning and skip those tests without failing if it can't find **xmllint**.

If you intend to use autoconf from Windows and you have Rational Rose installed, you may need to create */usr/bin/hostinfo* (inside of the Cygwin environment) as

```
#!/bin/false
```

so that **./configure**'s running of **hostinfo** doesn't run **hostinfo** from Rational Rose.

In order to use Visual C++ with abuild, you must have your environment set up to invoke Visual C++ command line tools. This can be achieved by running the shortcut supplied with Visual Studio, or you can create a batch file on your own. The following batch file would enable you to run abuild from a Cygwin environment with the environment set up for running Visual C++ from Visual Studio 7.1 (.NET 2003):

```
@echo off
call "%VS71COMNTOOLS%"\vsvars32.bat
C:\cygwin\cygwin.bat
```

Adjust as needed if your Cygwin is installed other than in *C:\cygwin* or you have a different version of Visual C++ installed.

In order to use qtest with abuild under Windows, the Cygwin version of Perl must be the first **perl** in your path.

2.5. Version Control Considerations

Abuild creates output directories in the source directory, and all generated files are created inside of these abuild-generated directories. All output directories are named *abuild-**. It is recommended that you configure hooks or triggers in your version control system to prevent these directories or their contents from being accidentally checked in. It may also be useful to prevent *Abuild.backing* from being checked in since this file always contains information about the

local configuration rather than something that would be CM controlled. If it is your policy to allow these to be checked in, they should be prevented from appearing in shared areas such as the trunk. [5]

[5] Note, however, that the abuild test suite contains *Abuild.backing* files, so any CM system that contains abuild must have an exception for abuild itself. It's conceivable that other tools could also have reasons to have checked in *Abuild.backing* files in test suites or as templates.

Chapter 3. Basic Operation

In this chapter, we will describe the basics of running abuild on a few simple build items, and we will describe how those build items are constructed. We will gloss over many details that will be covered later in the documentation. The goal of this chapter is to give you enough information to work on simple build items that belong to existing build trees. Definitions of *build item* and *build tree* appear below. More detailed information on them can be found in Chapter 4, *Build Items and Build Trees*, page 20. The examples we refer to in this chapter can be found in *doc/example/basic* in your abuild source or binary distribution.

3.1. System Considerations

Abuild imposes few system-based restrictions on how you set it up and use it, but here are a few important things to keep in mind:

- Avoid putting spaces in path names wherever possible. Although abuild tries to behave properly with respect to spaces in path names and is known to handle many cases correctly, make is notoriously bad at it. If you try to use spaces in path names, it is very likely that you will eventually run into problems as they generally cause trouble in a command-line environment.
- Be careful about the lengths of path names. Although abuild itself imposes no limits on this, you may run up against operating system limits if your paths are too long. In particular, Windows has a maximum path name length of 260 characters. If you have a build tree whose root already has a long path and you then have Java classes that are buried deep within a package-based directory structure, you can bump into the 260-character limit faster than you'd think. On Windows, it is recommended that you keep your build tree roots as close to the root of the drive as possible. On any modern UNIX system, you should not run into any path name length issues.

3.2. Basic Terminology

Here are a few basic terms you'll need to get started:

build item
: A *build item* is the most basic item that is built by abuild. It usually consists of a directory that contains files that are built. Any directory that contains an *Abuild.conf* file is a build item. We refer to the build item whose *Abuild.conf* resides in the current directory as the *current build item*.

build tree
: A *build tree* is a collection of build items arranged hierarchically in the file system. All build items in a build tree may refer to each other by name. Each build item knows the locations of its children within the file system hierarchy and the names of the build items on which it depends.

build forest
: A *build forest* is a collection of build trees. If there are multiple build trees in a forest, there may be one-way visibility relationships among the trees, which are declared similarly to dependency relationships among build items. We will return to this concept later in the documentation.

target
: A *target* is some specific product to be built. The term “target” means exactly the same thing with abuild as it does with other build systems such as make or ant. In fact, with the exception of a small handful of “special” targets, abuild simply passes any targets given to it onto the backend build system for processing. The most common targets are **all** and **clean**. For a more complete discussion of targets, see Section 9.1, “Build Targets”, page 38. Be careful not to confuse *target* with *target type*, defined in Section 5.1, “Platform Structure”, page 24.

For a more complete description of build items, build trees, and build forests, please see Chapter 4, *Build Items and Build Trees*, page 20.

3.3. Compiler Selection

Full details on compiler support and compiler selection are covered in Section 24.1, “Platform Selection”, page 155. To get started, on Linux systems, abuild will build with gcc by default. On Windows, if you run abuild from a shell that is appropriately set up to run Microsoft Visual C++ (as by following the command prompt shortcut provided as part of your Visual C++ implementation), abuild will automatically use Visual C++. If you have cygwin installed with gcc and the mingw runtime environment, abuild will attempt to use **gcc -mno-cygwin** to build as long as you set the *MINGW* environment variable to `1`, though bear in mind that abuild's mingw support is not entirely complete.

3.4. Building a C++ Library

The directory *cxx-library* under *doc/example/basic* contains a simple C++ library. Our library is called *basic-library*. It implements the single C++ class called ***BasicLibrary*** using the header file *BasicLibrary.hh* and the source file *BasicLibrary.cc*. Here are the contents of those files:

basic/cxx-library/BasicLibrary.hh

```
#ifndef __BASICLIBRARY_HH__
#define __BASICLIBRARY_HH__

class BasicLibrary
{
  public:
    BasicLibrary(int);
    void hello();

  private:
    int n;
};

#endif // __BASICLIBRARY_HH__
```

basic/cxx-library/BasicLibrary.cc

```
#include "BasicLibrary.hh"
#include <iostream>

BasicLibrary::BasicLibrary(int n) :
    n(n)
{
}

void
BasicLibrary::hello()
{
    std::cout << "Hello.  This is BasicLibrary(" << n << ")." << std::endl;
}
```

Building this library is quite straightforward. Abuild's build files are generally declarative in nature: they describe what needs to be done rather than how it is done. Building a C or C++ library is a simple matter of creating an *Abuild.mk*

file that describes what the names of the library targets are and what each library's sources are, and then tells abuild to build the targets using the C and C++ rules. Here is this library's *Abuild.mk* file:

basic/cxx-library/Abuild.mk

```
TARGETS_lib := basic-library
SRCS_lib_basic-library := BasicLibrary.cc
RULES := ccxx
```

The string *ccxx* as the value of the *RULES* variable indicates that this is C or C++ code ("c" or "cxx"). In order for abuild to actually build this item, we also need to create an *Abuild.conf* file for it. The existence of this file is what makes this into a build item. We present the file here:

basic/cxx-library/Abuild.conf

```
name: cxx-library
platform-types: native
```

In this file, the **name** key is used to specify the name of the build item and the **platform-types** key is used to help abuild figure out on which platforms it should attempt to build this item. Finally, we want this build item to be able to make the resulting library and header file available to other build items. This is done in its *Abuild.interface* file:

basic/cxx-library/Abuild.interface

```
INCLUDES = .
LIBDIRS = $(ABUILD_OUTPUT_DIR)
LIBS = basic-library
```

This tells abuild to add the directory containing this file to the include path, the output directory in which the generated targets were created to the library path, and the *basic-library* library to the list of libraries to be linked with. Notice that the name of the library assigned to the *LIBS* variable is the same as the value assigned to the *TARGETS_lib* variable in the *Abuild.mk* file, and that the abuild-provided variable *$(ABUILD_OUTPUT_DIR)* is used as the library directory. All relative paths specified in the *Abuild.interface* file are relative to the directory that contains the *Abuild.interface* file. They are automatically converted internally by abuild to absolute paths, which helps to keep build items location-independent.

To build this item, you would run the command **abuild** in the *basic/cxx-library* directory. Abuild will create an output directory whose name would start with *abuild-* and be based on the platform or platforms on which abuild was building this item. This is the directory to which the variable *$(ABUILD_OUTPUT_DIR)* refers in the *Abuild.interface* file.

There is a lot of capability hiding beneath the surface here and quite a bit of flexibility in the exact way in which this can be done, but this is the basic pattern you will observe for the majority of C and C++ library build items.

3.5. Building a C++ Program

The directory *basic/cxx-program* contains a simple C++ program. This program links against the library created in our previous example. Here is the main body of our program:

basic/cxx-program/program.cc

```
#include <BasicLibrary.hh>
```

```
int main()
{
    BasicLibrary b(5);
    b.hello();
    return 0;
}
```

This program includes the *BasicLibrary.hh* header file from the ***cxx-library*** build item. Here is the *Abuild.mk* for this build item:

basic/cxx-program/Abuild.mk

```
TARGETS_bin := cxx-program
SRCS_bin_cxx-program := program.cc
RULES := ccxx
```

Notice that this is very similar to the *Abuild.mk* from the library build item. The only real difference is that the *TARGETS* and *SRCS* variables contain the word *bin* instead of *lib*. This tells abuild that these are executable targets rather than library targets. Notice the conspicuous lack of any references to the library build item or the location of the headers or libraries that it makes available. A principal feature of abuild is that this program build item does not need to know that information. Instead, it merely declares a dependency on the ***cxx-library*** build item by name. This is done in its *Abuild.conf*:

basic/cxx-program/Abuild.conf

```
name: cxx-program
platform-types: native
deps: cxx-library
```

Notice the addition of the **deps** key in this file. This tells abuild that our program build item *depends* on the library build item. When abuild sees this, it automatically makes all the information in ***cxx-library***'s *Abuild.interface* available to ***cxx-program***'s build, alleviating the need for the ***cxx-program*** build item to know the locations of these files. This will also tell abuild that ***cxx-library*** must be built before we can build ***cxx-program***.

To build this item, we just run the **abuild** command as we did for ***cxx-library***. This will automatically build dependency ***cxx-library*** before building ***cxx-program***. In this way, you can can start a build from any build item and let abuild automatically take care of building all of its dependencies in the correct order.

The output of running **abuild** in the *cxx-program* directory when starting from a clean build is shown below. Your actual output will differ slightly from this. In particular, the output below has the string `--topdir--` in place of the path to *doc/example*, and the string `<native>` in place of your native platform. [1] Notice that abuild builds ***cxx-library*** first and then ***cxx-program***:

basic-cxx-program.out

```
abuild: build starting
abuild: cxx-library (abuild-<native>): all
make: Entering directory `--topdir--/basic/cxx-library/abuild-<native>'
Compiling ../BasicLibrary.cc as C++
```

[1] All example output in this document is normalized this way since it all comes directly from abuild's test suite. Testing all the examples in the test suite guarantees the accuracy of the examples and ensures that they work as advertised on all platforms for which abuild is released. Should you wish to study abuild's test suite with the examples, be aware that the bold italicized text preceding each block of example output is the name of the expected output file from the test suite.

```
Creating basic-library library
make: Leaving directory `--topdir--/basic/cxx-library/abuild-<native>'
abuild: cxx-program (abuild-<native>): all
make: Entering directory `--topdir--/basic/cxx-program/abuild-<native>'
Compiling ../program.cc as C++
Creating cxx-program executable
make: Leaving directory `--topdir--/basic/cxx-program/abuild-<native>'
abuild: build complete
```

To remove all of the files that abuild created in any build item's directory, you can run **abuild clean** in that directory. To clean everything in the build tree, run **abuild --clean=all**. More details of how to specify what to build and what to clean can be found in Chapter 9, *Telling Abuild What to Build*, page 38.

3.6. Building a Java Library

In our next example, we'll demonstrate how to build a simple Java library. You will find the Java example in *basic/java-library*. The files here are analogous to those in our C++ library example. First, here is a Java implementation of our ***BasicLibrary*** class:

basic/java-library/src/java/com/example/basic/BasicLibrary.java

```
package com.example.basic;

public class BasicLibrary
{
    private int n;

    public BasicLibrary(int n)
    {
        this.n = n;
    }

    public void hello()
    {
        System.out.println("Hello.  This is BasicLibrary(" + n + ").");
    }
}
```

Next, look at *Abuild.conf*:

basic/java-library/Abuild.conf

```
name: java-library
platform-types: java
```

This is essentially identical to our C++ library except that the **platform-types** key has the value **java** instead of the value **native**. This is always true for Java build items. Next, we'll look at the *Abuild.groovy* file:

basic/java-library/Abuild.groovy

```
parameters {
    java.jarName = 'java-library.jar'
```

```
    abuild.rules = 'java'
}
```

Java build items have this file instead of *Abuild.mk*. The contents are very similar. The *Abuild.groovy* file contains Groovy code that is executed inside a particular context provided by abuild. Most *Abuild.groovy* files will simply set *parameters* that describe what will be built. In this file, we set the *java.jarName* parameter to the name of the JAR file we are creating, and we set the *abuild.rules* parameter to the value `'java'` to indicate that we are using the *java* rules. For Java build items, we don't explicitly list the source files. Instead abuild automatically finds sources in a source directory which is, by default, *src/java*. There are many more parameters that can be set, and you have considerable flexibility about how to arrange things and how to get files into your Java archives. Abuild aims to allow you to *build by convention*, but it gives you the flexibility to do things your own way when you want to. We provide detailed information about the directory structure for Java builds in Section 19.3, “Directory Structure for Java Builds”, page 107.

Finally, look at the *Abuild.interface* file. This file provides information to other build items about what they should add to their classpaths in order to make use of the JAR file created by this build item:

basic/java-library/Abuild.interface

```
declare java-library.archiveName string = java-library.jar
declare java-library.archivePath filename = \
   $(ABUILD_OUTPUT_DIR)/dist/$(java-library.archiveName)
abuild.classpath = $(java-library.archivePath)
abuild.classpath.manifest = $(java-library.archivePath)
```

You'll notice here that we are actually setting four different variables. Not all of these are required, but the pattern here is one that you may well wish to adopt, especially if you are working in a Java Enterprise environment. The first statement in the interface file declares a variable called *java-library.archiveName* as a string and initializes it to the value `java-library.jar`. This syntax of declaring and initializing an interface variable was introduced into abuild with version 1.1. Here we adopt a convention of using the build item name as the first field of the variable name, and the literal string *archiveName* as the second field. By including the name of the build item in the name of the interface variable, we reduce the possibility of creating a name clash. By providing a variable to hold the name of the archive provided by this build item, we allow other build items to refer to this JAR file by name without having to know what it is called. The second interface variable, *java-library.archivePath*, contains the full path to the archive. (Notice that abuild puts the JAR file in the *dist* subdirectory of the abuild output directory.) This enables other build items to refer to this archive by path without knowing any details beyond this naming convention and the name of the providing build item. Making this type of information available in this way is not necessarily a straight Java “SE” environment, but it can be very useful in a Java “EE” environment where build items that create EAR files may have to reach into other build items to package their artifacts in higher level archives. Experience has shown that adopting a convention like this and following it consistently will pay dividends in the end.

After setting these two build-item-specific variables, we assign to two built-in variables: *abuild.classpath*, and *abuild.classpath.manifest*. Most simple JAR-providing build items will do this. Abuild actually provides multiple classpath variables, each of which is intended to be used in a particular way. For a discussion, please see Section 17.5.3, “Interface Variables for Java Items”, page 93.

As with the C++ library, it is possible to build this item by running **abuild** from the *basic/java-library* directory.

3.7. Building a Java Program

In Java, there is no deep distinction between a “library” and a “program” except that a JAR file that provides a program must have a *main* method. If a JAR file contains a main method, it can be executed, though it can also be used as a library. Here are the relevant files for the program example:

basic/java-program/src/java/com/example/basic/BasicProgram.java

```
package com.example.basic;

import com.example.basic.BasicLibrary;

public class BasicProgram
{
    public static void main(String[] args)
    {
        BasicLibrary l = new BasicLibrary(10);
        l.hello();
    }
};
```

basic/java-program/Abuild.conf

```
name: java-program
platform-types: java
deps: java-library
```

basic/java-program/Abuild.groovy

```
parameters {
    java.jarName = 'java-program.jar'
    java.mainClass = 'com.example.basic.BasicProgram'
    java.wrapperName = 'java-program'
    abuild.rules = 'java'
}
```

A JAR file's manifest file may identify a class that contains a *main* method. Abuild adds the **Main-Class** attribute to the manifest file when the *java.mainClass* parameter is set in the *Abuild.groovy*. In addition, abuild will create a wrapper script if the *java.wrapperName* parameter is set. The wrapper script that abuild creates may be useful for casual execution of the Java program for testing purposes, but it is generally not a substitution for having your own deployment mechanism. In particular, the wrapper script references items from your classpath by their paths within the build structure, and additionally, abuild's wrapper scripts are not as portable as the Java code that they help to invoke. [2]

Here is the output of running **abuild** in this directory. As in the C++ program example, the output has been modified slightly: in addition to the `--topdir--` substitution, we have also filtered out time stamps and other strings that could potentially differ between platforms:

basic-java-program.out

```
abuild: build starting
abuild: java-library (abuild-java): all
    [mkdir] Created dir: --topdir--/basic/java-library/abuild-java/classes
    [javac] Compiling 1 source file to --topdir--/basic/java-library/abu\
\ild-java/classes
```

[2] Specifically, abuild generates different wrapper scripts depending on whether you're running on Windows or not. Although it would work to build Java code on UNIX and run it on Windows, or vice versa, wrapper scripts generated on one platform are not portable to the other.

```
    [mkdir] Created dir: --topdir--/basic/java-library/abuild-java/dist
      [jar] Building jar: --topdir--/basic/java-library/abuild-java/dist\
\/java-library.jar
abuild: java-program (abuild-java): all
    [mkdir] Created dir: --topdir--/basic/java-program/abuild-java/classes
    [javac] Compiling 1 source file to --topdir--/basic/java-program/abu\
\ild-java/classes
    [mkdir] Created dir: --topdir--/basic/java-program/abuild-java/dist
      [jar] Building jar: --topdir--/basic/java-program/abuild-java/dist\
\/java-program.jar
abuild: build complete
```

Part II. Normal Operation

In this part of the manual, we discuss the standard features of abuild. For most ordinary build problems, these chapters provide all the information you will need. A few advanced topics are presented here. Where appropriate, they include cross references to later parts of the document where functionality is covered in more depth. By the end of this part, you should have a reasonably complete understanding of the structure of abuild's build trees, and a fairly complete picture of abuild's overall functionality. You will know enough about abuild to be able to use it for tasks of moderate complexity.

Chapter 4. Build Items and Build Trees

Now that we've had a chance to see abuild in action for a simple case, it's time to go into more detail about how things fit together. In Section 3.2, “Basic Terminology”, page 11, we briefly defined the terms *build item*, *build tree*, and *build forest*. In this chapter, we will describe them in bit more detail and briefly introduce a number of concepts that apply to them.

4.1. Build Items as Objects

A precise definition of *build item* would state that a build item is any directory that contains an *Abuild.conf*. Perhaps a more useful definition would say that a build item is the basic object that participates in abuild's object-oriented view of a software build. A build item provides some *service* within a build tree. Most build items build some kind of code: most often a library, executable, or Java archive. Build items may provide other kinds of services as well. For example, a build item may implement a code generator, support for a new compiler, or the ability to make use of a third-party software library. In addition, a build item may have certain attributes including a list of *dependencies*, a list of *supported flags*, information about what types of platforms the build item may be built on, a list of *traits*, and other non-dependency relationships to other build items. Each of these concepts is explored in more depth later in the document.

All build items that provide a service are required to have a name. Build item names must be unique within their build tree and all other build trees accessible to their build tree since the build item name is how abuild addresses a build item. Build item names consist of period-separated segments. Each segment may contain mixed case alphanumeric characters, underscores, and dashes. Build item names are case-sensitive.

The primary mechanism for describing build items is the *Abuild.conf* file. This file consists of colon-separated key/value pairs. A complete description of the *Abuild.conf* file may be found in Chapter 15, *The Abuild.conf File*, page 79. In the mean time, we will introduce keys as they become relevant to our discussion.

4.2. Build Item Files

Although every build item has an *Abuild.conf* file, there are various other files that a build item may have. We defer a complete list and detailed discussion these files for later in the document, but we touch briefly upon a few of the common ones here.

Abuild.conf
: This is the most basic of the build item files, and it is the only file that must be present for every build item. We sometimes refer to this as the *build item configuration file*.

Abuild.mk, *Abuild.groovy*
: These are the files that direct abuild what to actually build in a given build item. Each build file is associated with a specific backend. Exactly one of these files must be present in order for abuild to attempt to build a build item. As such, these files are known as *build files*. When we say that a build item has or does not have a build file, we are specifically talking about one of these files. In particular, it is important to note that *Abuild.conf* and *Abuild.interface* are not considered build files. [1]

Abuild.interface
: The *Abuild.interface* file is present for every build item that wants to make some product of its build accessible to other build items. We refer to this as the build item's *interface file*. There has been some confusion among some

[1] Additionally, the files *Abuild-ant.properties* and *Abuild-ant.xml* are recognized as build files, associated with the deprecated xml-based ant backend.

abuild users about the term *interface*. Please understand that abuild interfaces are distinct from Java interfaces, C++ header files, and so forth, though they serve essentially the same function. If you view a build item as an object, the abuild interface contains information about what services that object provides. It exposes the interfaces through which other build items will access a given build item's products.

4.3. Build Trees

A build tree, as defined before, is a collection of build items arranged hierarchically in the file system. Like build items, build trees have names, and are only referred to from other build trees by name. The root of a build tree is a build item whose *Abuild.conf* contains the **tree-name** key. We refer to this item as the tree's *root build item*.

A build tree is formed as a result of the items it contains holding references to the locations of their children within the file system hierarchy. These locations are named as relative paths in the **child-dirs** keys of the items' *Abuild.conf* files. It is customary to have the value of **child-dirs** contain single path elements (*i.e.*just a directory without any subdirectories), but this is also not a requirement: **child-dirs** entries may contain multiple path elements as long as there are no *Abuild.conf* files in any of the intermediate directories. If a build item's child contains its own **tree-name** key, that child build item is the root of a separate build tree that is part of the same forest, defined below. Otherwise, the child build item is part of the same tree as its parent.

In addition to containing build items, build trees can contain other attributes. Among these are references to other build trees, a list of *supported traits*, and a list of *plugins*. We will discuss these topics later in the document. These attributes are defined using keys in the root build item's *Abuild.conf* file.

4.4. Build Forests

A build forest is a collection of build trees that are connected to each other by virtue of one tree's root build item being referenced as a child of a build item in another tree in the forest. When abuild starts up, it looks for an *Abuild.conf* in the current directory. It then walks up the file system one directory at a time looking for additional *Abuild.conf* files. Eventually, it will either find the topmost *Abuild.conf* file, or it will find an *Abuild.conf* file that is not listed as a child of the next higher one. Whichever of these cases is found first, the resulting *Abuild.conf* file is the root of the build forest. The forest then consists of all the trees encountered by following all the **child-dirs** pointers from the forest root.

Note that, unlike with build items and trees, forests do not have names. Note also that, unlike with trees, there is no explicit marker of the root of a build forest. This is very important as it allows you to extend a forest from above without modifying the forest itself. For a more in-depth discussion, see Chapter 7, *Multiple Build Trees*, page 33.

Note that the hierarchy defined by the layout of build items in the file system is a file system hierarchy and nothing more. It doesn't have to have any bearing at all on the dependency relationships among the build items. That said, it is sensible to organize build items in a manner that relates to the architecture of the system, and this in turn usually has implications about dependencies. Still, it is important to keep in mind that abuild is not file-system driven but rather is dependency driven.

4.5. Special Types of Build Items

In further describing build items and their attributes, it is useful to classify build items into several types. Most build items serve the purpose of providing code to be compiled. There are a number of special types of build items that serve other purposes. We discuss these here:

root
: The root build item of a build tree is the topmost item in that tree. It has a **tree-name** key that gives the name of the build tree. It is often the case that the root build item serves no purpose other than to hold onto tree-wide attributes.

It is therefore permissible for a root build item to lack a **name** key. (See below for a discussion of unnamed build items.) Keys that define attributes of the build tree may appear only in the root build item's *Abuild.conf*.

unnamed

In order to refer to one build item from another, both build items must have names. Abuild requires that every named build item in a build forest be named uniquely within that forest. A name is given to a build item by setting the **name** key in its *Abuild.conf*. Sometimes, a build item exists for the sole purpose of bridging its parent with its children in the file system. Such items do not need to be referenced by other build items, so they do not need names. The only use of an unnamed build item is to serve as an intermediary during traversal of the file system. Such a build item's *Abuild.conf* may only contain the **child-dirs** key. Abuild doesn't retain any information about these build items. It simply traverses through them when locating build items at startup time. Unnamed build items are the only types of build items that don't have to belong to any particular build tree. It is common for the root of a forest to be an unnamed build item whose children are all roots of build trees.

interface-only

Interface-only build items are build items that contain (in addition to *Abuild.conf*) an *Abuild.interface* file. They do not build anything and therefore do not contain build files (such as *Abuild.mk* or *Abuild.groovy*). Since they have nothing to build, abuild never actually invokes a backend on them. They are, however, included in all dependency and integrity checks. A typical use of interface-only build items would be to add the locations of external libraries to the include and library paths (or to the classpaths for Java items). There may also be some interface-only build items that consist solely of static files (templated C++ classes, lists of constants, etc.). Interface-only build items may also be used to declare interface variables that are used by other build items.

pass-through

Pass-through build items are useful for solving certain advanced abuild problems. As such, there are aspects of this definition that may not be clear on the first reading. Pass-through build items contain no build or interface files, but they are named and have dependencies. This makes pass-through build items useful as top-level facades for hiding more complicated build item structures. This could include build items that have private names relative to the pass-through item, and it could also include structures containing build items that cross language and platform boundaries. Several examples in the documentation use pass-through build items to hide private build item names. For further discussion of using pass-through build items in a cross-platform environment, please see Section 24.4, “Dependencies and Pass-through Build Items”, page 159.

plugin

Plugins are capable of extending the functionality of abuild beyond what can be accomplished in regular build items. Plugins must be named and not have any dependencies. No other build items may depend on them. Plugins are a topic in their own right. They are discussed in depth in Chapter 29, *Enhancing Abuild with Plugins*, page 185.

4.6. Integrating with Third-Party Software

Virtually every software development project has some need to integrate with third-party software libraries. In a traditional build system, you might list the include paths, libraries, and library directories right in your *Makefile*, *build.xml*, or configuration file for whatever build system you are using. With abuild, the best way to integrate with a third-party library is to use a build item whose sole purpose is to export that library's information using an *Abuild.interface* file. In the simplest cases, a third-party library build item might be an interface only build item (described above) that just includes the appropriate library directives in a static *Abuild.interface* file. For example, a build item that provides access to the PCRE (Perl-compatible regular expression) libraries on a Linux distribution that has them installed in the system's standard include path might just include an *Abuild.interface* with the following contents:

```
LIBS = pcrecpp pcre
```

For Java build items, a third-party JAR build item would typically append the path to the JAR file to the *abuild.classpath.external* interface variable. (For a discussion of the various classpath variables, see Section 17.5.3, “Interface Variables for Java Items”, page 93.)

Sometimes, the process may be more involved. For example, on a UNIX system, it is often desirable to use autoconf to determine what interface is required for a particular library. We present an example of using autoconf with abuild in Section 18.3, “Autoconf Example”, page 99. Still other libraries may use pkg-config. For those libraries, it may make sense to create a simple set of build rules that automatically generate an *Abuild.interface after-build* file (also discussed in Section 18.3, “Autoconf Example”, page 99) by running the **pkg-config** command. An example pkg-config build item may be found in the *abuild-contrib* package available at abuild's web site [http://www.abuild.org].

Whichever way you do it for a given package, the idea is that you should always create a build item whose job it is to provide the glue between abuild and the third-party library. Other build items that need to use the third-party library can then just declare a dependency on the build item that provides the third-party library's interface. This simplifies the process of using third-party libraries and makes it possible to create a uniform standard for doing so within any specific abuild build tree. It also alleviates the need to duplicate information about the third-party library throughout your source tree. *Whenever you are duplicating knowledge about the path of some entity, you would probably be better off creating a separate build item to encapsulate that knowledge.*

Chapter 5. Target Types, Platform Types, and Platforms

Abuild was designed with multiplatform operation in mind from the beginning. Up to this point, we have largely glossed over how abuild deals with multiple platforms. In this chapter, we will cover this aspect of abuild's operation in detail.

5.1. Platform Structure

Abuild classifies platforms into a three-level hierarchy. The three levels are described by the following terms:

target type
: A *target type* encompasses the overall kind of targets that are being built. A target type essentially encapsulates a build paradigm. Abuild understands three target types: `platform-independent` for truly platform-independent products like scripts and documentation, `object-code` for compiled object code like C and C++, and `java` for Java byte code and related products. One could argue that Java code is platform-independent, but since Java code has its own build paradigm, abuild considers it to be a separate target type. Be careful not to confuse *target type* with *target*, defined in Section 3.2, “Basic Terminology”, page 11.

platform type
: A *platform type* essentially defines a grouping of platforms. Platform types belong to target types and contain platforms. When configuring build items, developers assign build items to platform types rather than to platforms or target types. The `platform-independent` target type has only platform type: `indep`. The `java` target type has only one platform type: `java`. [1] Platform types are most useful in the `object-code` target type. Abuild has only one built-in platform type in the `object-code` target type: `native`. The `native` platform type applies to build items that are expected to be able to be built and run on the host platform. Additional platform types to support embedded platforms or cross compilers can be added in plugins (see Section 29.3, “Adding Platform Types and Platforms”, page 186).

platform
: The abuild *platform* is the lowest level of detail in describing the environment in which a target is intended to be used. The expectation is that compiled products (object files, libraries, binary executables, java class files, etc.) produced for one platform are always compatible with other products produced for that platform but are not necessarily compatible with products produced for a different platform. If two different versions of a compiler generate incompatible object code (because of incompatible runtime library versions or different C++ name mangling conventions, for example), then a host running one compiler may generate output belonging to a different platform from the same host running a different version of the compiler. For the `indep` platform type in the `platform-independent` target type, there is only one platform, which has the same name as the platform type: `indep`. For the `java` platform type in the `java` target type, there is also only one platform, which also shares its name with the platform type: `java`. Platforms become interesting within the `object-code` target type. When we refer to platforms, we are almost always talking about `object-code` platforms.

This table (Table 5.1, “Built-in Platforms, Platform Types, and Target Types”, page 25) shows the target types along with the built-in platform types and platforms that belong to them.

[1] At one time, it was planned for abuild to support different platform types for different versions of Java byte code. Although this would have been useful for build trees that had complex requirements for mixing JDKs of different versions, this capability would have added a lot of complexity to support a practice that is unusual and largely undesirable.

Table 5.1. Built-in Platforms, Platform Types, and Target Types

Target Type	Platform Type	Platform
`object-code`	`native`	based on available tools
`java`	`java`	`java`
`platform-independent`	`indep`	`indep`

When a build item is defined with multiple platform types, they must all belong to the same target type. (Since the only target type that has more than one platform type is `object-code`, this means the target type of a build item with multiple platform types will always be `object-code`.) Some interface variables are also based on target type. For example, it may be permissible for a java build item to depend on a C++ build item if the C++ build item exports native code or provides an executable code generator, but it would never make sense for a java build item to have an include path or library path in the sense of a C/C++ build item. When one build item depends on another, the platforms on which the two build items are being built come into play. We discuss this in Chapter 24, *Cross-Platform Support*, page 155.

5.2. Object-Code Platforms

For target type `object-code`, platform identifiers are of the form `os.cpu.toolset.compiler`[`.option`], described below. In all cases, each field of the platform identifier must consist only of lower-case letters, numbers, dash, or underscore. The fields of the platform identifier are as follows:

os
: A broad description of the operating system, such as `linux`, `solaris`, `windows`, `cygwin`, or `vxworks`

cpu
: A CPU type identifier such as `ix86`, `x86_64`, `ppc`, `ppc64`, or `sparc`.

toolset
: A user-defined label for a collection of tools. This is a convenience field to separate things like different versions of compilers or runtime libraries. It can be set to any string, at which point the user is responsible for ensuring that it does in fact define a meaningful collection of tools. By default, abuild will create a toolset name based on the operating system distribution or similar factors. Examples include `rhel4` on a Red Hat Enterprise Linux 4 system, or `deb5` on a Debian GNU/Linux 5.x system. [2]

compiler
: An identifier for the compiler C/C++ compiler toolchain to be used. Abuild has built-in support for gcc on UNIX systems and for Microsoft Visual C++ and mingw on Windows systems. Users can provide their own compiler toolchains in addition to these. The mechanism for adding new compilers is described in Section 29.3, “Adding Platform Types and Platforms”, page 186.

option
: An optional field that is used to pass additional information to the GNU Make code that implements support for the compiler. Typical uses for options would be to define different debugging, profiling, or optimization levels.

All of the fields of the platform identifier are made available in separate variables within the interface parsing system. In addition, for `object-code` build items, the make variable *$(CCXX_TOOLCHAIN)* is set to the value of the compiler field. Here are some example platform identifiers:

[2] At present, it is possible to add new toolsets easily with plugins, but the only way to *override* the built-in default toolset would be to edit *private/bin/get_native_platform_data*, the perl script abuild uses to determine this information at startup. This may be addressed in a future version of abuild.

linux.ppc64.proj1default.gcc
linux.ppc64.proj1default.gcc.release
linux.ppc64.proj1default.gcc.debug
linux.x86.fc5.gcc
linux.x86.fc5.gcc.release
linux.x86.fc5.gcc.debug
windows.ix86.nt5.msvc
windows.ix86.cygwin-nt5.mingw
vxworks.pc604.windriver.vxgcc

5.3. Output Directories

When abuild builds an item, it creates an output directory under that item's directory for every platform on which that item is built. The output directory is of the form *abuild-`platform-name`*. Abuild itself and all abuild-supplied rules create files only inside of abuild output directories. [3]

When abuild invokes make, it always does so from an output directory. This is true even for platform-independent build items. In this way, even temporary files created by compilers or other build systems will not appear in the build item's local directory. This makes it possible to build a specific item for multiple platforms in parallel without having to be concerned about the separate builds overwriting each other's files.

When abuild builds items using the Groovy backend (and also using the deprecated xml-based ant backend), it performs those builds inside a single Java virtual machine instance. As such, it does not change its working directory to the output directory. (Java does not support changing current directories, and besides, there could be multiple builds going on simultaneously in different threads.) However, each Java-based build has its own private ant ***Project*** whose *basedir* property is set to the output directory. As such, all well-behaved ant tasks will only create files in the output directory.

[3] Abuild considers any directory whose name starts with *abuild-* and which contains a file named *.abuild* to be an output directory.

Chapter 6. Build Item Dependencies

Management of dependencies among build items is central to abuild's functionality. We have already gotten a taste of this capability in the basic examples included in Chapter 3, *Basic Operation*, page 11. In this chapter, we will examine dependencies in more depth.

6.1. Direct and Indirect Dependencies

The sole mechanism for declaring dependencies among build items in abuild is the **deps** key in a build item's *Abuild.conf*. Suppose build item ***A*** declares build item ***B*** as a dependency. The following line would appear in ***A***'s *Abuild.conf*:

```
deps: B
```

This declaration causes two things to happen:

- It ensures that ***B*** will be built before ***A***.
- It enables ***A*** to see all of the variable declarations and assignments in ***B***'s *Abuild.interface* file.

We illustrate both of these principles later in this chapter. For an in-depth discussion of build ordering and dependency-aware builds, see Chapter 9, *Telling Abuild What to Build*, page 38. For an in-depth discussion of abuild's interface system, see Chapter 17, *The Abuild Interface System*, page 83.

Another very important point about dependencies in abuild is that they are *transitive*. In other words, if ***A*** depends on ***B*** and ***B*** depends on ***C***, then ***A*** also implicitly depends on ***C***. This means that the conditions above apply to ***A*** and ***C***. That is, ***C*** is built before ***A*** (which it would be anyway since it is built before ***B*** and ***B*** is built before ***A***), and ***A*** sees ***C***'s interface in addition to seeing ***B***'s interface. [1] Assuming that ***A*** does not explicitly list ***C*** in its **deps** key, we would call ***B*** a *direct dependency* of ***A*** and ***C*** an *indirect dependency* of ***A***. We also say that build item dependencies are *inherited* when we wish to refer to the fact that build ordering and interface visibility are influenced by both direct and indirect dependencies.

Abuild performs various validations on dependencies. The most important of these is that no cyclic dependencies are permitted. [2] In other words, if ***A*** depends on ***B*** either directly or indirectly, then ***B*** cannot depend on ***A*** directly or indirectly. There are other dependency validations which are discussed in various places throughout this document.

By default, any build item can depend on any other build item by name. Abuild offers two mechanisms to restrict which items can depend on which other items. One mechanism is through build item name scoping rules, discussed below. The other mechanism is through use of multiple build trees, discussed in Chapter 7, *Multiple Build Trees*, page 33.

6.2. Build Order

Abuild makes no specific commitments about the order in which items will be built except that no item is ever built before its dependencies are built. The exact order in which build items are built, other than that dependencies are built before items that depend on them, should be considered an implementation detail and not relied upon. When abuild is invoked in with multiple threads (using the **--jobs** option, as discussed in Chapter 13, *Command-Line Reference*, page

[1] In fact, since ***B*** depends on ***C***, ***C***'s interface is effectively included as part of ***B***'s interface. This makes ***C***'s interface visible to all build items that depend on ***B***. The exact mechanism by which this works is described in Chapter 17, *The Abuild Interface System*, page 83.
[2] Stated formally, abuild requires that build item dependencies form a directed acyclic graph.

70), it may build multiple items in parallel. Even in this mode, abuild will never start building one build item until all of its dependencies have been built successfully.

6.3. Build Item Name Scoping

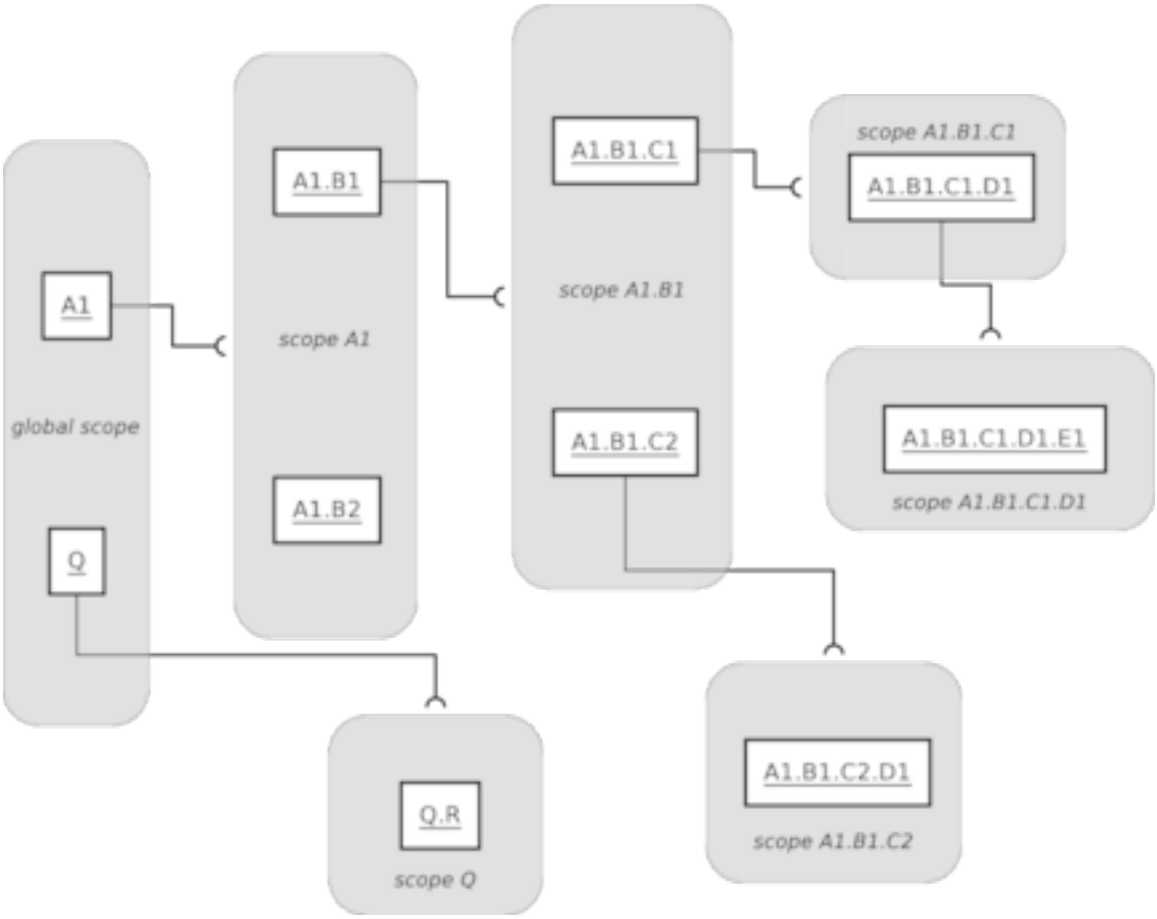
In this section, we discuss build item name scoping rules. Build item name scoping is one mechanism that can be used to restrict which build items may directly depend on which other build items.

Build item names consist of period-separated segments. The period separator in a build item's name is a namespace scope delimiter that is used to determine which build items may directly refer to which other build items in their *Abuild.conf* files. It is a useful mechanism for allowing a build item to hide the fact that it is composed of lower-level build items by blocking others from accessing those lower-level items directly.

Each build item belongs to a namespace scope equal to the name of the build item after removing the last period and everything following it. For example, the build item "***A.B.C.D***" is in the scope called "***A.B.C***". We would consider "***A.B***" and "***A***" to be *ancestor scopes*. The build item name itself also defines a scope. In this case, the scope "***A.B.C.D***" would contain "***A.B.C.D.E***". Any build item name scope that starts with "***A.B.C.D.***" (including the period) would be a *descendant scope* to "***A.B.C.D***". Any build item whose name does not contain a period is considered to belong to the global scope and is accessible by all build items.

One build item is allowed to access another build item by name if the referenced build item belongs to the accessing build item's scope or any of its ancestor scopes. Figure 6.1, "Build Item Scopes", page 29 shows a number of build items arranged by scope. In this figure, each build item defines a scope whose members appear in a gray box at the end of a semicircular arrowhead originating from the defining build item. Each build item in this figure can see the build items that are direct members of the scope that it defines, the build items that are siblings to it in its own scope, and the build items inside of any of its ancestor scopes. You may wish to study the figure while you follow along with the text below.

Figure 6.1. Build Item Scopes



Build items are shown here grouped by scope. Each build item is connected to the scope that it defines.

To illustrate, we will consider item ***A1.B1.C1***. The build item ***A1.B1.C1*** can access the following items for the following reasons:

- ***A1.B1.C1.D1*** because it belongs to the scope that ***A1.B1.C1*** defines: ***A1.B1.C1***
- ***A1.B1.C2*** because it is in the same scope as ***A1.B1.C1***: ***A1.B1***
- ***A1.B1*** and ***A1.B2*** because they belong to an ancestor scope: ***A1***
- ***A1*** and ***Q*** because they are global

It cannot access these items:

- ***A1.B1.C1.D1.E1*** because it is hidden in scope ***A1.B1.C1.D1***
- ***A1.B1.C2.D1*** because it is hidden in scope ***A1.B1.C2***
- ***Q.R*** because it is hidden in scope ***Q***

The item ***A1.B1.C1*** can *be accessed by* the following items:

- ***A1.B1*** because it is its parent

- ***A1.B1.C2*** because it is its sibling
- ***A1.B1.C1.D1*** and ***A1.B1.C1.D1.E1*** because they are its descendants
- ***A1.B1.C2.D1*** because it can see ***A1.B1.C1*** as a member of its ancestor scope ***A1.B1***

It cannot be accessed by these items:

- ***A1.B2***, ***A1***, ***Q***, and ***Q.R***, none of which can see inside of ***A1.B1***

To give a more concrete example, suppose you have a globally accessible build item called ***networking*** that was internally divided into private build items ***networking.src*** and ***networking.test***. A separate build item called ***logger*** would be permitted to declare a dependency on ***networking*** but not on ***networking.src*** or ***networking.test***. Assuming that it did not create any circular dependencies, ***networking.test*** would also be allowed to depend on ***logger***.

Note that these restrictions apply only to explicitly declared dependencies. It is common practice to implement a "public" build item as multiple "private" build items. The public build item itself would not have an *Abuild.interface* file, but would instead depend on whichever of its own private build items contain interfaces it wants to export. It would, in fact, be a pass-through build item. Because dependencies are inherited, items that depend on the public build item will see the interfaces of those private build items even though they would not be able to depend on them directly. In this way, the public build item becomes a facade for the private build items that actually do the work. For example, the build item ***networking*** would most likely not have its own *Abuild.interface* or *Abuild.mk* files. Instead, it might depend on ***networking.src*** which would have those files. It would probably not depend on ***networking.test*** since ***networking.test*** doesn't have to be built in order to use ***networking***. [3] This means that it would be okay for ***networking.test*** to depend on ***networking*** since doing so would not create any circular dependencies. Then, any build items that depend on ***networking*** indirectly depend on ***networking.src*** and would see ***networking.src***'s *Abuild.interface*.

There is nothing that a build item can do to allow itself to declare a direct dependency on another build item that is hidden within another scope: the only way to gain direct access to a build item is to be its ancestor or to be a descendant of its parent. (There are no restrictions on indirect access.) There are times, however, when it is desirable for a build item to allow itself to *be seen* by build items who would ordinarily not have access to it. This is accomplished by using the **visible-to** key in *Abuild.conf*. We defer discussion of this feature until later; see Chapter 25, *Build Item Visibility*, page 166.

6.4. Simple Build Tree Example

Now that the topic of build items and build trees has been explored in somewhat more depth, let's take a look at a simple but complete build tree. The build tree in *doc/example/general/reference/common* illustrates many of the concepts described above.

The first file to look at is the *Abuild.conf* belonging to this tree's root build item:

general/reference/common/Abuild.conf

```
tree-name: common
child-dirs: lib1 lib2 lib3
supported-traits: tester
```

[3] Although ***networking*** doesn't have to depend on ***networking.test***, you might be tempted to put the dependency in so that when you run the **check** target for all dependencies of ***networking***, you would get the test suite implemented in ***networking.test***. Rather than using a dependency for this purpose, you can use a trait instead. For information about traits, see Section 9.5, "Traits", page 42. A specific example of using traits for this purpose appears in that section.

This is a root build item configuration file, as you can see by the presence of the **tree-name** key. Notice that it lacks a **name** key, as is often the case with the root build item. This *Abuild.conf* contains the names of some child directories and also a build tree attribute: **supported-traits**, which lists the traits that are allowed in the build tree. We will return to the topic of traits in Section 9.5, “Traits”, page 42. In the mean time, we will direct our focus to the child build items.

The first child of the root build item of this tree is in the *lib1* directory. We examine its *Abuild.conf*:

general/reference/common/lib1/Abuild.conf

```
name: common-lib1
child-dirs: src test
deps: common-lib1.src
```

This build item is called ***common-lib1***. Notice that the name of the build item is not the same as the name of the directory, but it is based on the name of the directory. This is a typical strategy for naming build items. Abuild doesn't care how you name build items as long as they conform to the syntactic restrictions and are unique within a build tree. Coming up with a naming structure that parallels your system's architecture is a good way to help ensure that you do not create conflicting build item names. However, you should avoid creating build item names that slavishly follow your directory structure since doing so will make it needlessly difficult for you to move things around. A major feature of abuild is that nothing cares where a build item is located, so don't set a trap for yourself in which you have to rename a build item when you move it!

This build item does not have any build or interface files. It is a *pass-through build item*. It declares a single dependency: ***common-lib1.src***, and two child directories: *src* and *test*.

Next, look at the ***common-lib1.src*** build item's *Abuild.conf* in the *common/lib1/src* directory:

general/reference/common/lib1/src/Abuild.conf

```
name: common-lib1.src
platform-types: native
```

The first thing to notice is this build item's name. It contains a period and is therefore private to the ***common-lib1*** scope. That means that it is not accessible to build items whose names are not also under that scope. In particular, a build item called ***common-lib2*** would not be able to depend directly on ***common-lib1.src***. It would instead depend on ***common-lib1*** and would inherit the dependency on ***common-lib1.src*** indirectly.

This build item doesn't list any child directories and, as such, is a leaf in the file system hierarchy. It also happens not to declare any dependencies, so it is also a leaf in the dependency tree, though one does not imply the other. This build item configuration file contains the **platform-types** key, as is required for all build items that contain build or interface files. In addition to the *Abuild.conf* file, we have an *Abuild.mk* file and an *Abuild.interface* file:

general/reference/common/lib1/src/Abuild.mk

```
TARGETS_lib := common-lib1
SRCS_lib_common-lib1 := CommonLib1.cpp
RULES := ccxx
```

general/reference/common/lib1/src/Abuild.interface

```
INCLUDES = ../include
```

```
LIBDIRS = $(ABUILD_OUTPUT_DIR)
LIBS = common-lib1
```

There is nothing in these files that is fundamentally different from the basic C++ library example shown in Section 3.4, “Building a C++ Library”, page 12. We can observe, however, that the *INCLUDES* variable in *Abuild.interface* actually points to *../include* rather than the current directory. This simply illustrates that abuild doesn't impose any restrictions on how you might want to lay out your build items, though it is recommended that you pick a consistent way and stick with it for any given build tree. You should also avoid paths that point into other build items. Instead, depend on the other item and put the variable there. As a rule, if you ever have two interface variables or assignments that resolve to the same path, you are probably doing something wrong: a significant feature of abuild is that allows you to encapsulate the location of any given thing in only one place. Instead, figure out who *owns* a given file or directory and export it from that build item's interface. We will not study the source and header files in this example here, but you are encouraged to go to the *doc/example/general/reference/common* directory in your abuild source tree or installation directory to study the files further on your own.

Next, look at the *test* directory. Here is its *Abuild.conf*:

general/reference/common/lib1/test/Abuild.conf

```
name: common-lib1.test
platform-types: native
deps: common-lib1
traits: tester -item=common-lib1.src
```

Notice that it declares a dependency on ***common-lib1***. Since its name is also private to the ***common-lib1*** scope, it would have been okay for it to declare a dependency directly on ***common-lib1.src***. Declaring its dependency on ***common-lib1*** instead means that this test code is guaranteed to see the same interfaces as would be seen by any outside user of ***common-lib1***. This may be appropriate in some cases and not in others, but it demonstrates that it is okay for a build item that is inside of a particular namespace scope to depend on its parent in the namespace hierarchy. This build item also declares a trait, but we will revisit this when we discuss traits later in the document (see Section 9.5, “Traits”, page 42).

In addition to the *lib1* directory, we also have *lib2* and *lib3*. These are set up analogously to *lib1*, so we will not inspect every file. We will draw your attention to one file in particular: observe that the ***common-lib2.src*** build item in *reference/common/lib2/src* declares a dependency on ***common-lib3***:

general/reference/common/lib2/src/Abuild.conf

```
name: common-lib2.src
platform-types: native
deps: common-lib3
```

We will return to this build tree later to study build sets, traits, and examples of various ways to run builds.

Chapter 7. Multiple Build Trees

In large development environments, it is common to have collections of code that may be shared across multiple projects, and it's also common to have multiple development efforts being worked in parallel with the intention of integrating them at a later date. Ideally, such collections of shared code should be accessible by multiple projects but should not be able to access code from the those projects, and parallel development efforts should be kept independent to the maximum possible extent. In order to support this distributed and parallel style of software development, abuild allows you to divide your work up into multiple build trees, which coexist in a *build forest*. These trees can remain completely independent from each other, and you can also establish one-way dependency relationships among trees.

We define the following additional terms:

local build tree
: The *local build tree* is the build tree that contains the current directory.

tree dependency
: A *tree dependency* is a separate build tree whose items can supplement the local build tree. Build items in the local build tree can resolve the names of build items in the tree named as a tree dependency, but build items in the dependency cannot see items in the dependent (local) build tree.

top-level *Abuild.conf*
: The *top-level Abuild.conf* is an *Abuild.conf* file that is higher in th file system than any other *Abuild.conf* file in the build forest. If you are building a single tree, the top-level *Abuild.conf* file is typically the root build item of that tree. If you are building multiple trees, you have to create a higher-level *Abuild.conf* file that can reach the roots of all the trees you are going to use, directly or indirectly, through its **child-dirs** key.

7.1. Using Tree Dependencies

Even when abuild knows about multiple trees, it still won't allow items in one build tree to refer to items in other trees without an explicit instruction to do so. This makes it possible to ensure that items in one tree are not *accidentally* modified to depend on items in a tree that is supposed to be unrelated. When you want items in one tree to be able to use items in another tree, you declare a *tree dependency* of one tree on another. This creates a one-way relationship between the two trees such that items in the *dependent* tree (the one that declares the dependency) can see items in the tree on which it depends, but no visibility is possible in the other direction. To declare a tree dependency, you list the name of the tree dependency in the **tree-deps** key of the dependent tree's *Abuild.conf* file. As with item dependencies listed in **deps**, abuild requires that there are no cycles among tree dependencies.

There is nothing special about a build tree that makes it able to be the target of a tree dependency: any tree can depend on any other tree as long as no dependency cycles are created.

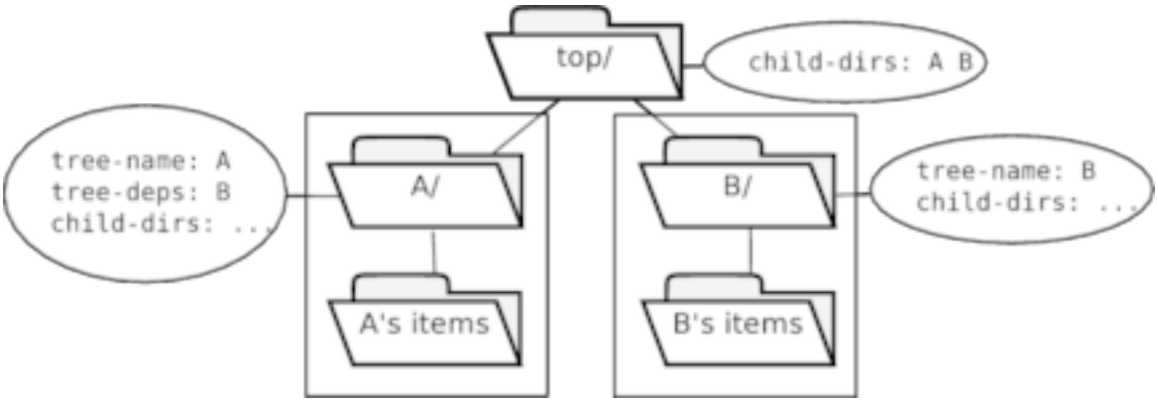
Once you set up another tree as a tree dependency of your tree, all build items defined in the tree named by the tree dependency are available to you (subject to normal scoping rules) as if they were in your local build tree. Since any tree can potentially have a dependency relationship with any other, abuild enforces that none of the build items in any build tree may have the same name as any build item in any tree in the forest. In order to avoid build item name clashes, it's a good idea to pick a naming convention for your build items that includes some kind of tree-based prefix, as we have done with names like ***common-lib1***.

7.2. Top-Level *Abuild.conf*

When you declare another tree as a tree dependency of your tree, you declare your dependency on the other tree by mentioning its name in the **tree-deps** of your tree's root *Abuild.conf*. In order for this dependency to work, abuild must

know where to find the tree. Abuild finds items and trees in the same way: it traverses the build forest from the top down and creates a table mapping names to paths. If the tree your tree depends on is *inside* of your tree, this poses no problem. But what if it is an external tree that is not inside your tree? In this instance, you must place the external tree somewhere within your overall build area, such as in another subdirectory of the parent of your own tree's root. Then you must create an *Abuild.conf* file in that common parent directory that knows about the root directories of the two trees. This is illustrated in Figure 7.1, “Top-Level *Abuild.conf*”, page 34.

Figure 7.1. Top-Level *Abuild.conf*



Tree *A* declares a tree dependency on tree *B*. In order for *A* to find *B*, an *Abuild.conf* file that points to both trees' locations must be created in a common ancestor directory. The ovals show the contents of each directory's *Abuild.conf* files.

The tree named *B* has an *Abuild.conf* that that declares no tree dependencies. It is a self-contained tree. However, *A*'s *Abuild.conf* file mentions *B by name*. How does *A* find *B*? When you start abuild, it walks up the tree to find the highest-level *Abuild.conf* (or the highest level one not referenced as a child of the next higher *Abuild.conf*) and traverses downward from there. In this case, the *Abuild.conf* in *A*'s parent directory knows the locations of both *A* and *B*. In this way, abuild has figured out where to find *B* when *A* declares the tree dependency. This is illustrated with a concrete example below.

7.3. Tree Dependency Example

In order for abuild to use multiple trees, it must be able to find the roots of all the trees when it traverses the file system looking for *Abuild.conf* files. As described earlier, abuild locates the root of the forest by looking up toward the root of the file system for other *Abuild.conf* files that list previous *Abuild.conf* directories in their **child-dirs** key. The parent directory of our previous example contains (see Section 6.4, “Simple Build Tree Example”, page 30) the following *Abuild.conf* file:

general/reference/Abuild.conf

```
child-dirs: common project derived
```

This is an unnamed build item containing only a **child-dirs** key. The **child-dirs** key lists not only the *common* directory, which is the root of the *common* tree, but also two other directories: *project* and *derived*, each of which we will discuss below. These directories contain additional build tree root build items, thus making them known to any abuild invocation that builds *common*. It is also okay to create one build tree underneath another named tree. As with build items, having one tree physically located beneath another doesn't have any implications about the dependency relationships among the trees.

We will examine a new build tree that declares the build tree from our previous example as an dependency. This new tree, which we will call the project build tree, can be found at *doc/example/general/reference/project*. The first file we examine is the new build tree's root build item's *Abuild.conf*:

general/reference/project/Abuild.conf

```
tree-name: project
tree-deps: common
child-dirs: main lib
```

This build item configuration file, in addition to having the **tree-name** key (indicating that it is a root build item), also has a **tree-deps** key, whose value is the word *common*, which is the name of the tree whose items we want to use. Note that, as with build items, abuild never requires you to know the location of a build tree.

Inside the project build tree, the ***project-lib*** build item is defined inside the *lib* directory. It is set up exactly the same way as ***common-lib1*** and the other libraries in the *common* tree. Here is its *Abuild.conf*:

general/reference/project/lib/Abuild.conf

```
name: project-lib
child-dirs: src test
deps: project-lib.src
```

Now look at ***project-lib.src***'s *Abuild.conf*:

general/reference/project/lib/src/Abuild.conf

```
name: project-lib.src
platform-types: native
deps: common-lib1
```

Notice that it declares a dependency on ***common-lib1***, which is defined in the *common* tree. This works because abuild automatically makes available to you all the build items in any build trees your depends on.

This build tree also includes a main program, but we will not go through the rest of the files in depth. You are encouraged to study the files on your own. There are also examples of traits in this build tree. We will return to this build tree during our discussion of traits (see Section 9.5, “Traits”, page 42).

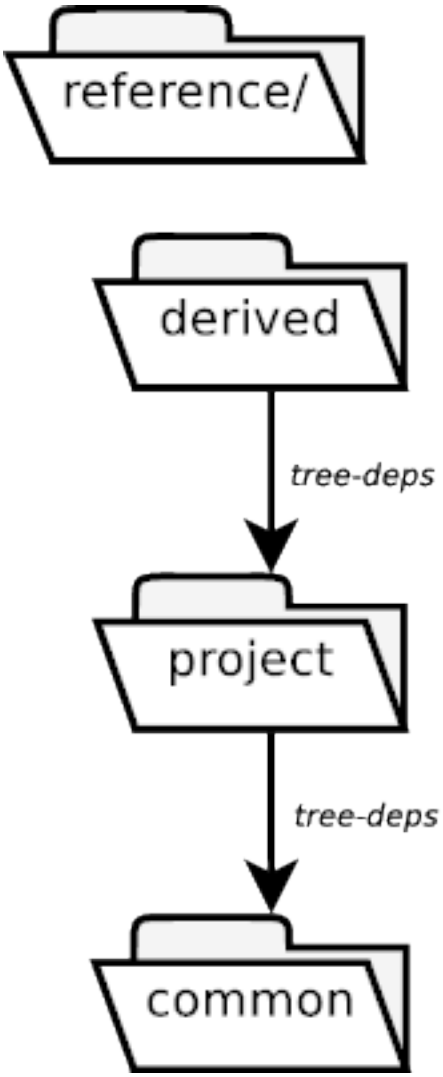
When you declare another build tree as a tree dependency, you automatically inherit any tree dependencies that *that* tree declared, so like item dependencies, tree dependencies are transitive. If this were not the case, abuild would not be able to resolve dependencies declared in the other tree if those dependencies were resolved in one of *its* tree dependencies. To illustrate this, we have a third build tree located in *doc/example/general/reference/derived*. This build tree is for a second project that is derived from the first project. This build tree declares *project* as an tree dependency as you can see in its root *Abuild.conf* file:

general/reference/derived/Abuild.conf

```
tree-name: derived
tree-deps: project
child-dirs: main
```

For a diagram of the entire *general/reference* collection of build tress, see Figure 7.2, “Build Trees in *general/reference*”, page 36.

Figure 7.2. Build Trees in *general/reference*



The *derived* build tree declares a dependency on the *project* build tree. The *project* build tree declares a dependency on the *common* build tree.

The *derived* build tree contains a ***derived-main*** build item structured identically to the C++ program build items we've seen earlier. Here at the main program's *Abuild.conf*:

general/reference/derived/main/src/Abuild.conf

```
name: derived-main.src
platform-types: native
deps: common-lib2 project-lib
traits: tester -item=derived-main.src
```

In this file, you can see that ***derived-main.src*** depends on ***project-lib*** from the *project* build tree and also ***common-lib2*** which is found in *project*'s dependency, *common*. We will return to this build tree in the examples at the end of Chapter 9, *Telling Abuild What to Build*, page 38.

Chapter 8. Help System

Abuild has a built-in help system, introduced in version 1.1, that makes it easier for users to get help for abuild itself and also for available rules, both built-in and user-supplied. All help text that is part of the abuild distribution can also be seen in Appendix E, *Online Help Files*, page 270.

The starting point to abuild's help system is the command **abuild --help**. Help is available on a variety of general topics including the help system and command invocation. You can also get help on rules. You can see information about what kinds of help is available on rules by running **abuild --help rules**.

The rules help facility offers three major capabilities. By running **abuild --help rules list**, you can see the list of compiler toolchains and also the list of available rules that you can assign to *RULES* (make) or *abuild.rules* (Groovy). In addition to telling you what's offered overall, this will tell you what target types the rules apply to, and whether the rules are available to you through your dependency chain. That way, if you need to make use of a rule that is provided by some build item that you don't depend on, you can know which item you need to add a dependency on to gain access to the rule. Once you know which toolchain or rule you want help on, you can use **abuild --help rules toolchain:*`toolchain-name`*** or **abuild --help rules rule:*`rule-name`*** to get available help for that toolchain or rule.

Creating help files is very straightforward. For any toolchain support file or rule file, in the same directory, create a text file called *`toolchain-name`-help.txt* or *`rule-name`-help.txt* as appropriate. The contents of this help file will be displayed to the user when help is requested on that toolchain or rule. Lines within the help text that start with "#" are ignored, which makes it possible for you to include notes to people who might be maintaining the help file. Also, abuild normalizes line terminators, displaying the help with whatever the platform's native line terminator is.

We present examples of help files in this manual as we present information about adding rules and toolchain support files. You can also run **abuild --help helpfiles** for a reminder about the help file format. (This text is also available in Section E.2, "**abuild --help helpfiles**", page 271.) To see an example of rule help, see Section 22.2, "Code Generator Example for Make", page 130. To see an example of toolchain help, see Section 29.5.3, "Platforms and Platform Type Plugins", page 194.

Chapter 9. Telling Abuild What to Build

Up to this point, we have seen only simple invocations of abuild to build a single item with all of its dependencies. Abuild offers several ways of creating sets of build items to build or clean. These are known as *build sets*. In addition, abuild's list of items to build can be expanded or restricted based on *traits* that are assigned to build items.

9.1. Build Targets

As defined in Section 3.2, “Basic Terminology”, page 11, the term *target* refers to a specific build product. In most cases, abuild passes any targets specified on the command line to the backend build system. Abuild provides several standard targets (see Chapter 13, *Command-Line Reference*, page 70). We have already encountered **all** and **clean** in earlier examples. It is also possible to add new targets through mechanisms that are covered later in the document. For now, you really only need to know a few things about targets:

- Different targets tell abuild to build different things.
- The **all** target is abuild's default target. When abuild builds a build item in order to satisfy a dependency, building the **all** target is required to be sufficient to satisfy the needs of items that depend on it. This means that the **all** target is responsible for building all parts of a build item that are potentially needed by any of its dependencies. This may seem significant, but it's a detail that takes care of itself most of the time.
- With the exception of two *special targets*, abuild doesn't do anything itself with targets other than pass them onto the backend build tool.

Abuild defines two *special targets*: **clean** and **no-op**. These targets are special in two ways: abuild does not allow them to be combined with other targets, and abuild handles them itself without passing them to a backend.

The **clean** target is used to remove the artifacts that are built by the other targets. Abuild implements the **clean** target by simply removing all abuild-generated output directories (see Section 5.3, “Output Directories”, page 26). When abuild processes the **clean** target, it ignores any dependency relationships among build items. Since it ignores dependencies and performs the cleanup itself without invoking a backend, running the **clean** target or cleaning multiple items using a clean set (described below) is very fast.

Note that, starting with version 1.0.3, abuild cleans *all* build items, not just those with build files. There are several reasons for this:

- In certain debugging modes, such as interface debugging mode, abuild may create output directories for items that don't build anything.
- You might change a build item from an item that builds something to an interface-only build item. In this case, you will want a subsequent clean to remove the no-longer-needed output directories.
- Although it is not necessarily recommended, there are some use cases in which build items may “push” files into the output directory of an interface-only build item. Some people may choose to implement installers that work this way. Having abuild clean interface-only build items makes it easier to clean up in those cases.

The **no-op** target is used primarily for debugging build tree problems. When abuild is invoked with the **no-op** target, it goes through all the motions of performing a build except that it does not read any *Abuild.interface* files or invoke any backends. It does, however, perform full validation of *Abuild.conf* files including dependency and integrity checking. This makes **abuild no-op**, especially with a build set (described below), very useful for taking a quick look at what items would be built on what platforms and in what order. We make heavy use of the **no-op** target in the examples at the end of this chapter so that we can illustrate certain aspects of build ordering without being concerned about the actual compilation steps.

9.2. Build Sets

We have already seen that, by default, abuild will build all of the build items on which the current item depends (directly or indirectly) in addition to building the current item. Now we generalize on this concept by introducing *build sets*. A build set is a collection of build items defined by certain criteria. Build sets can be used both to tell abuild which items to build and also to tell it which items to clean. [1] When abuild is invoked with no build set specified, its default behavior is to build all of the current item's dependencies as well as the current item. Sometimes, you may wish to assume all the dependencies are up to date and just build the current build item *without* building any of its dependencies. To do this, you may invoke abuild with the **--no-deps** option. This will generally only work if all dependencies are up to date. Using **--no-deps** is most convenient when you are in the midst of the edit/compile/test cycle on a single build item and you want to save the time of checking whether a potentially long chain of dependencies is already up to date. [2]

To instruct abuild to build all the items in a specific build set, run **abuild --build=*`set-name`*** (or **abuild -b *`set-name`***). To instruct abuild to clean all the items in a specific build set, run **abuild --clean=*`set-name`*** (or **abuild -c *`set-name`***). When building a build set, abuild will also automatically build any items that are direct or indirect dependencies of any items in the build set. However, if you specify any explicit targets on the command line, abuild will not, by default, apply those targets to items that it only added to the build set to satisfy dependencies; it will build those items with the **all** target instead. This is important as it enables you to add custom targets to a build item without necessarily having those targets be defined for build items it depends on. If you want abuild to build dependencies with explicitly named targets as well, use the **--apply-targets-to-deps** option. When cleaning with a build set, abuild does not ordinarily also clean the dependencies of the items in the set. To apply the **clean** target to all the dependencies as well, we also use the **--apply-targets-to-deps** option. This is a bit subtle, so we present several examples below.

The following build sets are defined:

current
: the current build item (*i.e.*, the build item whose *Abuild.conf* is in the current directory); abuild's default behavior is identical to **--build=current**

deps
: all direct and indirect dependencies of the current build item but not the item itself

desc
: all build items located at or below the current directory (items that are *desc*endants of the current directory)

descending
: alias for **desc**

down
: alias for **desc**

local
: all items in the build tree containing the item in the current directory; *i.e.*, the local build tree without any of its trees dependencies, noting that items in tree dependencies may, as always, still to be built to satisfy item dependencies

deptrees
: all items in the build tree containing the item in the current directory as well as all items in any of its tree dependencies [3]

[1] In retrospect, the term *build item set* would probably have been a better name for this. Just keep in mind that build sets can be used for both building and cleaning, and that when we use build sets for cleaning, we sometimes call them *clean sets* instead.
[2] In abuild 1.0, this was the default behavior, and the **--with-deps** option was required in order to tell abuild to build the dependencies.
[3] This is what the [all] build set did in abuild 1.0. In abuild 1.1, [all] may be more expansive since abuild now actually knows about all trees in the forest, not just those referenced by the current tree.

descdeptrees
all build items that are located at or below the current directory and are either in the current build tree or one of its dependencies—effectively the intersection between **desc** and **deptrees** [4]

all
all items in all known build trees, including those items in trees that are not related to the current build tree

name:*item-name[,item-name,...]*
all build items whose names are listed

pattern:*regular-expression*
all build items whose names match the given perl-compatible regular expression

Ordinarily, when you invoke **abuild clean** or **abuild --clean=*set-name***, abuild will remove all output directories for any affected build items. You may also restrict abuild to remove only specified output directories. There are two ways to do this. One way is to run **abuild clean** from inside an output directory. In that case, abuild will remove all the files in the output directory. [5] The other way is to use the **--clean-platforms** option, which may be followed by a shell-style regular expression that is matched against the platform portion of the output directory name. Examples are shown below.

9.2.1. Example Build Set Invocations

abuild
builds the **all** target for all dependencies of the current directory's build item and for the current directory; equivalent to **abuild --build=current**

abuild --no-deps
builds the current directory without building any of its dependencies

abuild check (or **abuild --build=current check**)
builds the **check** target for the current build item and the **all** target for all of its direct and indirect dependencies

abuild --apply-targets-to-deps check
builds the **check** target for the current build item and all of its direct and indirect dependencies

abuild --build=local check
builds the **check** target for all build items in the local build tree and the **all** target for any dependencies of any local items that may be satisfied in other trees

abuild --build=deptrees check
builds the **check** target for all build items in the local build tree and all of its tree dependencies

abuild --clean=current (or **abuild clean**)
removes all output directories for the current build item but not for any of its dependencies

abuild --clean=desc
removes all output directories for all build items at or below the current directory but not any of its dependencies

abuild --clean=all --clean-platforms java --clean-platforms '*.ix86.*'
for all build items, removes all *abuild-java* output directories and all output directories for platforms containing the string "*.ix86.*"

[4] This is what the [desc] build set did in abuild 1.0. In abuild 1.1, [desc] includes *all* build items at or below the current directory, but in abuild 1.0, abuild didn't know about those not in the dependency chain of the current tree. This build set is provided so there is an equivalent in abuild 1.1 to every build set from abuild 1.0. There are relatively few reasons to ever use it.
[5] In abuild 1.0, abuild actually passed the **clean** target to the backend, but abuild version 1.1 handles this **clean** invocation internally as it does for other **clean** invocations.

abuild --clean=current --apply-targets-to-deps
removes all output directories for the current build item and everything it depends on; useful when you want to try a completely clean build of a particular item

abuild --apply-targets-to-deps --clean=desc
removes all output directories for all build items at or below the current directory and all of their direct or indirect dependencies, including those that are not located at or below the current directory

abuild --build=name:lib1,lib2 xyz
builds the custom **xyz** target for the ***lib1*** and ***lib2*** build items and the **all** target for their direct or indirect dependencies

abuild --build=pattern:'.*\.test'
builds the **all** target for any item whose name ends with `.test` and any of those items' direct or indirect dependencies

abuild -b all
builds the **all** target for all build items in all known trees in the forest

abuild -c all
removes all output directories in all the build trees in the forest

9.3. Using build-also for Top-level Builds

Starting with abuild version 1.0.3, it is possible to list other build items in the **build-also** key of any named build item's *Abuild.conf* file. Starting with abuild version 1.1.4, **build-also** keys can list entire trees and also add options to include tree dependencies or other items at or below the item's directory. When abuild adds any build item to the build set, if that build item has a **build-also** key, then any build items listed there are also added to the build set. The operation of expanding initial build set membership using the **build-also** key is applied iteratively until no more build items are added. The principal intended use of this feature is to aid with setting up virtual “top-level” build items. For example, if your system consisted of multiple, independent subsystems and you wanted to build all of them, you could create a build item that lists the main items for each subsystem in a **build-also** key.

Arguments to **build-also** may be as follows:

[`item:`]*`item-name`* [**-desc**]
Add *`item-name`* to the build set. The literal `item:` prefix may be omitted for backward compatibility.

If the **-desc** option is given, all items at or below the directory containing *`item-name`* are also added to the build set. This is equivalent to running **abuild --build=desc** from *`item-name`*'s directory.

`tree:`*`tree-name`* [**-desc**] [**-with-tree-deps**]
If `tree:`*`tree-name`* is specified by itself, all items in the build tree named *`tree-name`* are added to the build set. This is equivalent to running **abuild --build=local** somewhere in that tree.

If **-desc** appears as an option by itself, all items at or below the directory containing the root of *`tree-name`* are added to the build set. This is equivalent to running **abuild --build=desc** from the directory containing the root of the tree.

If **-with-tree-deps** appears as an option by itself, all items in all trees that *`tree-name`* specifies as tree dependencies are added to the build set in addition to all items in *`tree-name`* itself. This is equivalent to running **abuild --build=deptrees** somewhere in that tree.

If **-with-tree-deps** and **-desc** are both specified, the result is to add the items that are in the *intersection* of the two options specified individually. In other words, all items that are in any dependent tree *and* are at or below the directory containing the root of the tree are added to the build set. This is equivalent to running **abuild --**

build=descdeptrees at the root of the build tree. Note that if you want the *union* of **-desc** and **-with-tree-deps** instead of the intersection, you simply have to specify both `tree:`*`tree-name`* **-desc** and `tree:`*`tree-name`* **-with-tree-deps** in the **build-also** key.

In older versions of abuild, the only way to force building of one build item to build another item was to declare dependencies or tree dependencies. This had several disadvantages, including the following:

- Adding unnecessary dependencies puts needless constraints on build ordering and parallelism.
- Using dependencies for this purpose is clumsy if there are multiple target types involved. It would require you to use a platform-specific dependency, which in turn could interfere with proper use of platform selectors.
- Otherwise harmless interface variable name clashes or assignment issues could cause problems as a result of having two interfaces that were supposed to be independent being loaded together.

Whenever you want building of one build item to result in building of another build item and the first item doesn't need to use anything from the items it causes to be built, it is appropriate to use **build-also** instead of a dependency.

9.4. Building Reverse Dependencies

Starting with abuild version 1.1, it is possible to use the **--with-rdeps** flag to instruct abuild to expand the build set by adding all *reverse dependencies* of any build item initially in the build set. When combined with **--repeat-expansion**, this process is applied iteratively so that all forward and reverse dependencies of every item in the build set will also be in the build set. [6] This can be especially useful if you are changing a widely used item and you want to make sure your change didn't break any build items that use your item. For additional details on how the build set is constructed, see Section 33.5, “Construction of the Build Set”, page 217..

9.5. Traits

In abuild, it is possible to assign certain traits to a build item. Traits are a very powerful feature of abuild. This material is somewhat more complicated than anything introduced up to this point, so don't worry if you have to read this section more than once.

Traits are used for two main purposes. Throughout this material, we will refer back to the two purposes. We will also provide clarifying examples later in the chapter.

The first purpose of traits is creation of semantically defined groups of build items. In this case, a trait corresponding to the grouping criteria would be applied to a build item directly. For example, all build items that can be deployed could be assigned the **deployable** trait.

A second purpose of traits is to create specific relationships among build items. These relationships may or may not correspond to dependencies among build items. These traits may be applied to a build item by itself or in reference to other build items. For example, the **tester** trait may be applied to a general system test build item by itself and may be applied to every test suite build item with a reference to the specific item being tested.

Traits are used to assist in the construction of build sets. In particular, you can narrow a build set by removing all items that don't have all of a specified list of traits. You can also expand a build set to add any build items that relate to any items already in the set by referring to them through all of a specified list of traits. This makes it possible to say things like “run the **deploy** target for every build item that has the **deployable** trait,” or “run the **test** target for every item that tests my local build item or anything it depends on.”

[6] Stated formally, when abuild is invoked with both **--with-rdeps** and **--repeat-expansion**, the build set is closed with respect to forward and reverse dependencies.

Since traits are visible in abuild's **--dump-data** output (see Appendix F, *--dump-data Format*, page 296), they are available to scripts or front ends to abuild. They may also be used for purely informational purposes such as specifying the classification level of a build item or applying a uniform label to all build items that belong to some group. Trait names are subject to the same constraints as build item names: they are case-sensitive and may consist of mixed case alphanumeric characters, numbers, underscores, dashes, and periods. Unlike with build items, the period does not have any semantic meaning when used in a trait name.

Starting with abuild 1.1.6, a build item that uses either the GNU Make backend or the Groovy backend (but not the deprecated xml-based ant backend) may also get access to the list of traits that are declared in its *Abuild.conf* file. For the GNU Make backend, the variable *ABUILD_TRAITS* contains a list of traits separated by spaces. For the Groovy backend, the variable *abuild.traits* contains a groovy list of traits represented as strings. In both cases, any information about referent build items is excluded; only the list of declared traits is provided. Possible uses for this information would include having a custom rule check to make sure a given trait is specified before providing a particular target, having it give an error if a particular trait is not defined, or even having it change behavior on the basis of a trait.

9.5.1. Declaring Traits

Any named build item may include a **traits** key that lists one or more of the traits that are supported in its build tree. The list of traits supported in a build tree is given as the value of the **supported-traits** key in the root build item's *Abuild.conf*. The list of supported traits is inherited through tree dependencies, so any trait declared as valid in any trees your tree depends on are also available. The set of traits that can be specified on the command line is the union of all traits allowed by all known trees.

Traits listed in the **traits** key can be made referent to other build items by listing the other build items in an **-item** option. For example, the following *Abuild.conf* fragment declares that the ***potato.test*** build item is deployable, unclassified, and a tester for the ***potato.lib*** and ***potato.bin*** build items:

```
this: potato.test
traits: deployable tester -item=potato.lib -item=potato.bin unclassified
```

9.5.2. Specifying Traits at Build Time

To modify the build set or clean set based on traits, use the **--only-with-traits** and **--related-by-traits** command-line options to abuild. These options must be combined with the specification of a build set. They correspond to the two purposes of traits discussed above.

To build all build items that have all of a specified list of traits, run **abuild --build=*set* --only-with-traits *trait*[,*trait*,...]**. This is particularly useful when semantically grouped build items share a common custom target. For example, if all the deployable build items had a special **deploy** target, you could run the **deploy** target for all deployable items in the local build tree with the command

```
abuild --build=local --only-with-traits deployable deploy
```

If multiple traits are specified at once, only build items with all of the specified traits are included.

Once a build set has been constructed, you may want to add additional items to the set based on traits. Specifically, you may want to add all items related by a trait to items already in the build set. To expand a build set in this way, run **abuild --build=*set* --related-by-traits *trait*[,*trait*,...]** For example, if you wanted to run the **test** target for all build items that are declared as testers (using the **tester** trait) of your build item or any of its dependencies, you could run the command

```
abuild --build=current --related-by-traits=tester test
```

As above, if multiple traits are specified at once, only build items that are related by all of the specified traits are included. Note that the same trait may be used referent to another build item or in isolation. The **--related-by-traits** option only applies to traits used in reference to other build items. For example, if a build item had the **tester** trait not referent to any build item, it would not be picked up by the above command. The **--only-with-traits** option picks up all build items that have the named traits either in isolation or referent to other build items.

It is also possible to combine these options. In that case, the build set is first restricted using **--only-with-traits** and then expanded using the **--related-by-traits** as shown in examples below. The order of the arguments has no effect on this behavior.

Ordinarily, when a specific target is specified as an argument to abuild (as in **abuild test** or **abuild deploy** rather than just **abuild**), abuild runs that target for every item initially in the build set (before dependency expansion). When the build set is expanded or restricted based on traits, any explicitly specified targets are run only for build items that have the specified traits. This is important because it enables you to use traits to group build items that define specific custom targets.

If **--related-by-traits** and **--only-with-traits** are both specified, any explicit targets are applied only to traits named in **--related-by-traits** as the effect of that option is applied last. All other build items are built with the **all** target. Note that the **--apply-targets-to-deps** option will cause any explicit targets to be applied to all build items, as always. Later in this chapter, we review the exact rules that abuild uses to decide which targets to apply to which build items.

The **--list-traits** flag provides information about which traits can be used on the command line. To see more detailed information about which traits were made available in which build trees, you can examine the output of **abuild --dump-data** (see Appendix F, *--dump-data Format*, page 296).

For more detailed information about how build sets are constructed with respect to traits, please see Section 33.5, “Construction of the Build Set”, page 217.

9.5.3. Example Trait Invocations

abuild --build=desc --only-with-traits deployable deploy
: Run the **deploy** target for all items at or below the current directory that have the **deployable** trait, and run the **all** target for all items that they depend on.

abuild --build=current --related-by-traits tester test
: Build the current build item and all of its dependencies with the **all** target, and run the **test** target for any build items that declared themselves as a tester for any of those items. Any additional dependencies of the testers would also be built with the **all** target.

abuild --build=local --only-with-traits deployable,tester deploy test
: Run both the **deploy** and the **test** targets for any build items in the local build tree (the current build item's tree excluding its tree dependencies) that have both the **deployable** and the **tester** traits either specified alone or in reference to other build items. Run the **all** target for their dependencies.

abuild --build=all --only-with-traits requires-hw --related-by-traits tester hwtest
: Run the **all** target for all items that have the **requires-hw** trait as well as any of their dependencies, and run the **hwtest** target for all items that test any of them. Additional dependencies of the testers would also be built with the **all** target.

9.6. Target Selection

Although we have described how various options affect which build items are built with which targets, we summarize that information here so that it all appears in one place. Put simply, the default behavior is that abuild applies any explicitly named targets to all build items that directly match the criteria for belonging to the named build set. Any

build items that abuild is building just to satisfy dependencies are built with the **all** target. This behavior is overridden by specifying **--apply-targets-to-deps**, which causes abuild to build all build items with the explicit targets. The exact rules are described in the list below. These rules apply only when a build set is specified with **--build** or **-b**. There are several mutually exclusive cases:

1. The **--apply-targets-to-deps** option was specified or the explicit target is **no-op**. In this case, any explicitly named targets are applied to all items in the build set.

2. The **--apply-targets-to-deps** option was not specified, the target is not **no-op**, and no trait arguments were specified. In this case, all items that were initially added to the build set, along with any build items specified by any of their **build-also** keys (with the **build-also** relationship applied recursively) are built with any explicitly specified targets. Any other build items added to the build set to satisfy dependencies are built with the **all** target.

3. The **--apply-targets-to-deps** option was not specified, the target is not **no-op**, **--only-with-traits** *was* specified, and **--related-by-traits** *was not* specified. In this case, all items belonging to the original build set (including **build-also** expansion) and having all of the named traits are built with the explicit targets. Other items (dependencies of build items with the named traits but that do not have the named traits themselves) are built with the **all** target.

4. The **--apply-targets-to-deps** option was not specified, the target is not **no-op**, and **--related-by-traits** *was* specified. In this case, the build set is first constructed normally and then restricted to any items that have all the traits specified in the **--only-with-traits** option, if any. Then it is expanded to include any build item related to one of the original build set members by all the traits named in **--related-by-traits**. These related items are built with the explicit targets. Other items, including additional dependencies of related items, are built with the **all** target.

For more detailed information on how the build set is constructed, please see Section 33.5, “Construction of the Build Set”, page 217.

9.7. Build Set and Trait Examples

Now that we've seen the topics of build sets and traits, we're ready to revisit our previous examples. This time, we will talk about how traits are used in a build tree, and we will demonstrate the results of running abuild with different build sets. We will also make use of the special target **no-op** which can be useful for debugging your build trees.

9.7.1. Common Code Area

Any arguments to abuild that are not command-line options are interpreted as targets. By default, abuild uses the **all** target to build each build item in the build set. If targets are named explicitly, for the build items to which they apply, they are passed directly to the backend. There are two exceptions to this rule: the special targets **clean** and **no-op** are trapped by abuild and handled separately without invocation of the backend. We have already seen the **clean** target: it just removes any abuild output directories in the build item directory. The special **no-op** target causes abuild to go through all the motions of building except for actually invoking the backend. The **no-op** command is useful for seeing what build items would be built on what platforms in a particular invocation of abuild. It does all the same validation on *Abuild.conf* files as a regular build, but it doesn't look at *Abuild.interface* files or build files (*Abuild.mk*, etc.).

We return now to the *reference/common* directory to demonstrate both the **no-op** target and some build sets. From the *reference/common* directory, we can run **abuild --build=local no-op** to tell abuild to run the special **no-op** target for every build item in the local build tree. Since this tree has no tree dependencies, there is no chance that there are any dependencies that are satisfied outside of the local build tree. Running this command produces the following results (with the native platform again replaced by the string `<native>`):

reference-common-no-op.out

```
abuild: build starting
```

```
abuild: common-lib1.src (abuild-<native>): no-op
abuild: common-lib1.test (abuild-<native>): no-op
abuild: common-lib3.src (abuild-<native>): no-op
abuild: common-lib2.src (abuild-<native>): no-op
abuild: common-lib2.test (abuild-<native>): no-op
abuild: common-lib3.test (abuild-<native>): no-op
abuild: build complete
```

Of particular interest here is the order in which abuild visited the items. Abuild makes no specific commitments about the order in which items will be built except that no item is ever built before its dependencies are built. [7] Since ***common-lib2.src*** depends on ***common-lib3.src*** (indirectly through its dependency on ***common-lib3***), abuild automatically builds ***common-lib3.src*** before it builds ***common-lib2.src***. On the other hand, since ***common-lib2.test*** has no dependency on ***common-lib3.test***, no specific ordering is necessary in that case. If you were to run **abuild --clean=local** from this directory, you would not observe the same ordering of build items since abuild does not pay any attention to dependencies when it is running the clean target, as shown:

reference-common-clean-local.out

```
abuild: cleaning common-lib1 in lib1
abuild: cleaning common-lib1.src in lib1/src
abuild: cleaning common-lib1.test in lib1/test
abuild: cleaning common-lib2 in lib2
abuild: cleaning common-lib2.src in lib2/src
abuild: cleaning common-lib2.test in lib2/test
abuild: cleaning common-lib3 in lib3
abuild: cleaning common-lib3.src in lib3/src
abuild: cleaning common-lib3.test in lib3/test
```

Note also that only the build items that have *Abuild.mk* files are cleaned. Abuild knows that there is nothing to build in items without *Abuild.mk* files and skips them when it is building or cleaning multiple items.

If you are following along, then go to the *reference/common* directory and run **abuild --build=desc check**. This will build and run the test suites for all build items at or below that directory, which in this case, is the same collection of build items as the **local** build set. [8] This produces the following output, again with some system-specific strings replaced with generic values:

reference-common-check.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib1/src/a\
\build-<native>'
Compiling ../CommonLib1.cpp as C++
Creating common-lib1 library
make: Leaving directory `--topdir--/general/reference/common/lib1/src/ab\
```

[7] In fact, when abuild creates a build order, it starts with a lexically sorted list of build trees and re-orders it as needed so that trees appear in dependency order. Then, within each tree, it does the same with items. The effect is that items build by tree with most referenced trees building earlier and, with each tree, most referenced items building earlier. Ties are resolved by lexical ordering. That said, the exact order of build items, other than that dependencies are built before items that depend on them, should be considered an implementation detail and not relied upon. Also, keep in mind that, in a multithreaded build, the order is not deterministic, other than that no item's build is started before all its dependencies' builds have completed.

[8] The test suites in this example are implemented with QTest [http://qtest.qbilt.org], which therefore must be installed for you to run them. See Chapter 10, *Integration with Automated Test Frameworks*, page 57.

```
\uild-<native>'
abuild: common-lib1.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib1/test/\
\abuild-<native>'
Compiling ../main.cpp as C++
Creating lib1_test executable

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/lib1.test
lib1  1 (test lib1 class)                                        ... PASSED

Overall test suite                                               ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/common/lib1/test/a\
\build-<native>'
abuild: common-lib3.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib3/src/a\
\build-<native>'
Compiling ../CommonLib3.cpp as C++
Creating common-lib3 library
make: Leaving directory `--topdir--/general/reference/common/lib3/src/ab\
\uild-<native>'
abuild: common-lib2.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib2/src/a\
\build-<native>'
Compiling ../CommonLib2.cpp as C++
Creating common-lib2 library
make: Leaving directory `--topdir--/general/reference/common/lib2/src/ab\
\uild-<native>'
abuild: common-lib2.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib2/test/\
\abuild-<native>'
Compiling ../main.cpp as C++
Creating lib2_test executable

********************************
STARTING TESTS on ---timestamp---
********************************
```

```
Running ../qtest/lib2.test
lib2  1 (test lib2 class)                                          ... PASSED

Overall test suite                                                 ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/common/lib2/test/a\
\build-<native>'
abuild: common-lib3.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib3/test/\
\abuild-<native>'
Compiling ../main.cpp as C++
Creating lib3_test executable

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/lib3.test
lib3  1 (test lib3 class)                                          ... PASSED

Overall test suite                                                 ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/common/lib3/test/a\
\build-<native>'
abuild: build complete
```

This example includes the output of qtest test suites. QTest is a simple and robust automated test framework that is integrated with abuild and used for abuild's own test suite. For information, see Section 10.2, “Integration with QTest”, page 57.

By default, when abuild builds multiple build items using a build set, it will stop after the first build failure. Sometimes, particularly when building a large build tree, you may want abuild to try to build as many build items as it can, continuing on failure. In this case, you may pass the **-k** option to abuild. When run with the **-k** option, abuild will

continue building other items after one item fails. It will also exit with an abnormal exit status after it builds everything that it can, and it will provide a summary of what failed. When run with **-k**, abuild also passes the corresponding flags to the backends so that they will try to build as much as they can without stopping on the first error. Both the make and Groovy backends behave similarly to abuild: they will keep going on failure, skip any targets that depend on failed targets, and exit abnormally if any failures are detected.

Ordinarily, if one build item fails, abuild will not attempt to build any other items that depend on the failed item even when run with **-k**. If you specify the **--no-dep-failures** option along with **-k**, then abuild will not only continue after the first failure but will also attempt to build items even when one or more of their dependencies have failed. Use of this option may result in cascading errors since the build of one item is likely to fail as a result of failures in its dependencies. There are, however, several cases in which this option may still be useful. For example, if building a large build tree with known problems in it, it may be useful to first tell abuild to build everything it possibly can. Then you can go back and try to clean up the error conditions without having to wait for the compilation of files that would have been buildable before. Another case in which this option may be useful is when running test suites: in many cases, we may wish to attempt to run test suites for items even if some of the test suites of their dependencies have failed. Essentially, running **-k --no-dep-failures** allows abuild to attempt to build everything that the backends will allow it to build.

9.7.2. Tree Dependency Example: Project Code Area

Returning to the project area, we demonstrate how item dependencies may be satisfied in trees named as tree dependencies and the effect this has on the build set. Under *reference/project*, we have just two public build items called ***project-main*** and ***project-lib***. The ***project-lib*** build item is structured like the libraries in the common area. The ***project-main*** build item has a *src* directory that builds an executable and has its own test suite. We have already seen that *reference/project/Abuild.conf* has a **tree-deps** key that lists *common* and that items from the *project* tree depend on build items from *common*. Specifically, ***project-lib*** depends on ***common-lib1*** and ***project-main*** depends on ***common-lib2*** which in turn depends on ***common-lib3***.

If we go to *reference/project/main/src* and run **abuild no-op**, we see the following output:

reference-project-main-no-op.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): no-op
abuild: common-lib3.src (abuild-<native>): no-op
abuild: common-lib2.src (abuild-<native>): no-op
abuild: project-lib.src (abuild-<native>): no-op
abuild: project-main.src (abuild-<native>): no-op
abuild: build complete
```

Notice here that abuild only built the build items whose names end with ***.src***, that it built the items in dependency order, and that it built all the items from *common* before any of the items in *project*. We can also run **abuild --apply-targets-to-deps check** to run the **check** target for each of these build items. This generates the following output:

reference-project-main-check.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): check
abuild: common-lib3.src (abuild-<native>): check
abuild: common-lib2.src (abuild-<native>): check
abuild: project-lib.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/project/lib/src/a\
```

```
\build-<native>'
Compiling ../ProjectLib.cpp as C++
Creating project-lib library
make: Leaving directory `--topdir--/general/reference/project/lib/src/ab\
\uild-<native>'
abuild: project-main.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/project/main/src/\
\abuild-<native>'
Compiling ../main.cpp as C++
Creating main executable

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/main.test
main  1 (testing project-main)                                    ... PASSED

Overall test suite                                                ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/project/main/src/a\
\build-<native>'
abuild: build complete
```

The presence of the **--apply-target-to-deps** flag caused the **check** target will be run for our dependencies as well as the current build item. In this case, there were no actions performed building the files in *common* because they were already built. If individual files had been modified in any of these build items, the appropriate targets would have been rebuilt subject to the ordinary file-based dependency management performed by make or ant.

9.7.3. Trait Example

In our previous example, we saw the **check** target run for each item (that has a build file). Since the items other than ***project-main*** don't contain their own test suites, we see the test suite only for ***project-main***. Sometimes we might like to run all the test suites of all the build items we depend on, even if we don't depend on their test suites directly. We can do this using traits, assuming our build tree has been set up to use traits for this purpose. Recall from earlier that our *common* build tree declared the **tester** trait in its root build item's *Abuild.conf*. Here is that file again:

general/reference/common/Abuild.conf

```
tree-name: common
child-dirs: lib1 lib2 lib3
supported-traits: tester
```

Also, recall that all the test suites declared themselves as testers of the items that they tested. Here again is ***common-lib1.test***'s *Abuild.conf*, which declares ***common-lib1.test*** to be a tester of ***common-lib1.src***:

general/reference/common/lib1/test/Abuild.conf

```
name: common-lib1.test
platform-types: native
deps: common-lib1
traits: tester -item=common-lib1.src
```

Given that all of our build items are set up in this way, we can instruct abuild to run the test suites for everything that we depend on. We do this by running **abuild --related-by-traits tester check**. This runs the **check** target for every item that declares itself as a tester of the current build item or any of its dependencies, and the **all** target for everything else, including any additional dependencies of any of those test suites. That command generates the following output:

reference-project-main-trait-test.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): all
abuild: common-lib1.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib1/test/\
\abuild-<native>'

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/lib1.test
lib1  1 (test lib1 class)                                      ... PASSED

Overall test suite                                             ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/common/lib1/test/a\
\build-<native>'
abuild: common-lib3.src (abuild-<native>): all
abuild: common-lib2.src (abuild-<native>): all
abuild: common-lib2.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib2/test/\
\abuild-<native>'

********************************
STARTING TESTS on ---timestamp---
```

```
********************************


Running ../qtest/lib2.test
lib2  1 (test lib2 class)                                          ... PASSED

Overall test suite                                                 ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/common/lib2/test/a\
\build-<native>'
abuild: common-lib3.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/common/lib3/test/\
\abuild-<native>'

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/lib3.test
lib3  1 (test lib3 class)                                          ... PASSED

Overall test suite                                                 ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/common/lib3/test/a\
\build-<native>'
abuild: project-lib.src (abuild-<native>): all
abuild: project-lib.test (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/project/lib/test/\
\abuild-<native>'
Compiling ../main.cpp as C++
Creating lib_test executable

********************************
```

```
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/lib.test
lib  1 (test lib class)                                          ... PASSED

Overall test suite                                               ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/project/lib/test/a\
\build-<native>'
abuild: project-main.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/project/main/src/\
\abuild-<native>'

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/main.test
main  1 (testing project-main)                                   ... PASSED

Overall test suite                                               ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/project/main/src/a\
\build-<native>'
abuild: build complete
```

Observe that the previously unbuilt ***project-lib.test*** build item was built using the **check** target by this command, and that all the test suites were run. If your development area has good test suites, you are encouraged to use a trait to indicate which items they test as we have done here using the **tester** trait. This enables you to run the test suites of items in your dependency chain. This can give you significant assurance that everything you depend on is working the way it is supposed to be each time you start a development or debugging session.

9.7.4. Building Reverse Dependencies

Suppose you have made a modification to a particular build item, and you want to make sure the modification doesn't break anyone who depends on that build item, whether the dependent item is in the modified item's tree or not. In order to do this, you can specify the **--with-rdeps** flag when building the modified item. This will cause abuild to add all of that item's reverse dependencies to the build set. For example, this is the output of running **abuild --with-rdeps** in the *general/reference/common/lib2/src* directory:

reference-common-lib2-rdeps.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): no-op
abuild: common-lib3.src (abuild-<native>): no-op
abuild: common-lib2.src (abuild-<native>): no-op
abuild: common-lib2.test (abuild-<native>): no-op
abuild: common-lib3.test (abuild-<native>): no-op
abuild: project-lib.src (abuild-<native>): no-op
abuild: project-main.src (abuild-<native>): no-op
abuild: derived-main.src (abuild-<native>): no-op
abuild: build complete
```

This includes all direct and indirect reverse dependencies of *common-lib2.src*. If you really want to be make sure that everything that is related to this build item by dependency in any way is rebuilt, you can use the **--repeat-expansion** option as well. This will repeat the reverse dependency expansion after adding the other dependencies of your reverse dependencies, and will continue repeating the expansion until no more items are added. If we run **abuild --with-rdeps --repeat-expansion no-op** from here, we get this output:

reference-common-lib2-rdeps-repeated.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): no-op
abuild: common-lib1.test (abuild-<native>): no-op
abuild: common-lib3.src (abuild-<native>): no-op
abuild: common-lib2.src (abuild-<native>): no-op
abuild: common-lib2.test (abuild-<native>): no-op
abuild: common-lib3.test (abuild-<native>): no-op
abuild: project-lib.src (abuild-<native>): no-op
abuild: project-lib.test (abuild-<native>): no-op
abuild: project-main.src (abuild-<native>): no-op
abuild: derived-main.src (abuild-<native>): no-op
abuild: build complete
```

Observe the addition of *common-lib1.test* and *project-lib.test*, which are reverse dependencies of libraries added to satisfy the dependencies of some of *common-lib2*'s dependencies! If that seems confusing, then you probably don't need to worry about ever using **--repeat-expansion**! Using **--repeat-expansion** with **--with-rdeps** will usually a lot of build items to the build set. In this example, it actually adds every build item in the forest to the build set. The only build items that would not be added would be completely independent sets of build items that happen to exist in the same forest.

9.7.5. Derived Project Example

Finally, we return to our derived project build tree in *reference/derived*. This build tree declares *project* as a tree dependency. As pointed out before, although *derived* does not declare *common* as a tree dependency, it can still use build

items in *common* because tree dependencies are transitive. If we run **abuild --build=desc check** from *reference/derived*, we will see all our dependencies in *common* and *project* being built (though all are up to date at this point) before our own test suite is run, and we will also see that all the items in *common* build first, followed by the items in *project*, finally followed by the items in *derived*. This is the case even though they are not all descendants of the current directory. This again illustrates how abuild adds additional items to the build set as required to satisfy dependencies:

reference-derived-check.out

```
abuild: build starting
abuild: common-lib1.src (abuild-<native>): all
abuild: common-lib3.src (abuild-<native>): all
abuild: common-lib2.src (abuild-<native>): all
abuild: project-lib.src (abuild-<native>): all
abuild: derived-main.src (abuild-<native>): check
make: Entering directory `--topdir--/general/reference/derived/main/src/\
\abuild-<native>'
Compiling ../main.cpp as C++
Creating main executable

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/main.test
main  1 (testing derived-main)                                  ... PASSED

Overall test suite                                              ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/reference/derived/main/src/a\
\build-<native>'
abuild: build complete
```

We can also observe that we do not see this behavior with the special **clean** target. Both **abuild --clean=desc** and **abuild --clean=local** produce this output when run from *reference/derived*:

reference-derived-clean-local.out

```
abuild: cleaning derived-main in main
abuild: cleaning derived-main.src in main/src
```

As another demonstration of the transitive nature of tree dependencies, run **abuild --clean=all** from the root of the *derived* build tree. That generates this output:

reference-derived-clean.out

```
abuild: cleaning common-lib1 in ../common/lib1
abuild: cleaning common-lib1.src in ../common/lib1/src
abuild: cleaning common-lib1.test in ../common/lib1/test
abuild: cleaning common-lib2 in ../common/lib2
abuild: cleaning common-lib2.src in ../common/lib2/src
abuild: cleaning common-lib2.test in ../common/lib2/test
abuild: cleaning common-lib3 in ../common/lib3
abuild: cleaning common-lib3.src in ../common/lib3/src
abuild: cleaning common-lib3.test in ../common/lib3/test
abuild: cleaning derived-main in main
abuild: cleaning derived-main.src in main/src
abuild: cleaning project-lib in ../project/lib
abuild: cleaning project-lib.src in ../project/lib/src
abuild: cleaning project-lib.test in ../project/lib/test
abuild: cleaning project-main in ../project/main
abuild: cleaning project-main.src in ../project/main/src
```

Here are a few things to notice:

- We clean all build items in *common* and *project* as well as in *derived*.
- Even build items that don't contain build files are visited.
- Build items are cleaned in an order that completely disregards any dependencies that may exist among them.

Chapter 10. Integration with Automated Test Frameworks

Abuild is integrated with two automated test frameworks: QTest, and JUnit. Additional integrations can be performed with plugins or build item rules or hooks.

10.1. Test Targets

Abuild defines three built-in targets for running test suites: **check**, **test**, and **test-only**. The **check** and **test** targets are synonyms. Both targets first ensure that a build item is built (by depending on the **all** target) and then run the build item's test suites, if any. The **test-only** target also runs a build item's test suite, but it does not depend on **all**. This means that it will almost certainly fail when run on a clean build tree. The **test-only** target is useful for times when you *know* that a build item is already built and you want to run the test suite on what is there *now*. One case in which you might want to do this would be if you had just started editing some source files and decided you wanted to rerun the test suite on the existing executables before rebuilding them. Another case in which this could be useful is if you had just built a build tree and then wanted to immediately go back and run all the test suites without having to pay the time penalty of checking to make sure each build is up to date. In this case, you could run **abuild --build=all test-only** immediately after the build was completed. Such a usage style might be appropriate for autobuilders or other systems that build and test a build tree in a controlled environment.

10.2. Integration with QTest

Abuild is integrated with the QTest [http://qtest.qbilt.org] test framework. The QTest framework is a perl-based test framework intended to support a *design for testability* testing mentality. Abuild's own test suite is implemented using QTest. When using either the make or the Groovy backends, if a directory called *qtest* exists, then the **test** and **check** targets will invoke **qtest-driver** to run qtest-based test suites. If a single file with the *.testcov* extension exists in the build item directory, abuild will invoke **qtest-driver** so that it can find the test coverage file and activate test coverage support. Note that abuild runs **qtest-driver** from the output directory, so the coverage output files as well as *qtest.log* and *qtest-results.xml* will appear in that directory. If you wish to have a qtest-based test suite be runnable on multiple platforms simultaneously, it's best to avoid creating temporary files in the *qtest* directory. If you wish to use the abuild output directory for your temporary files, you can retrieve the full path to this directory by calling the *get_start_dir* method of the qtest ***TestDriver*** object.

In order to use test coverage, you must add source files to the *TC_SRCS* variable in your *Abuild.mk* or *Abuild.groovy* file. Abuild automatically exports this into the environment. If you wish to specify a specific set of tests to run using the *TESTS* environment variable, you can pass it to abuild on the command line as a variable definition (as in **abuild check TESTS=some-test**), and abuild will automatically export it to the environment for qtest.

10.3. Integration with JUnit

When performing ant-based builds using the Groovy framework, if the *java.junitTestsuite* property is set to the name of a class, then the **test** and **check** targets will attempt to run a JUnit-based test suite. You can also set *java.junitBatchIncludes* to a pattern that matches a list of class files, in which case JUnit tests will be run from all matching classes. JUnit is not bundled with abuild, so it is the responsibility of the build tree maintainer to supply the required JUnit JARs to abuild. The easiest way to do this is to create a plugin that adds the JUnit JARs to *abuild.classpath.external* in a *plugin.interface* file. (For more details on plugins, please see Chapter 29, *Enhancing Abuild with Plugins*, page 185.) You can also copy the JAR file for a suitable version of JUnit into either ant's or abuild's *lib* directory, as any JAR files in those two locations are automatically added to the classpath.

10.4. Integration with Custom Test Frameworks

Adding support for your additional test frameworks is straightforward and can be done by creating a plugin that adds the appropriate code to the appropriate targets. For make-based items, you must make sure that your tests are run by the **check**, **test**, and **test-only** targets. You also must ensure that your **check** and **test** targets depend on **all** and that your **test-only** target does not depend on **all**. For Groovy-based items, you must make sure that your tests are run by the **test-only** target, and abuild will take care of making sure it is run by the **test** and **check** targets. For details on plugins, see Chapter 29, *Enhancing Abuild with Plugins*, page 185. For details on writing make rules, see Section 30.2, “Guidelines for Make Rule Authors”, page 205. For details on writing rules for the Groovy backend, see Section 30.3, “Guidelines for Groovy Target Authors”, page 206.

Chapter 11. Backing Areas

In a large development environment, it is very common for a developer to have a local work area that contains only the parts of the system that he or she is actually working on. Any additional parts of the software that are required in order to satisfy dependencies would be resolved through some kind of outside, read-only reference area. Abuild provides this functionality through the use of *backing areas*.

11.1. Setting Up Backing Areas

Backing areas operate at the build forest level. Any build forest can act as a backing area. If abuild needs to reference a build item that is found in the local forest, it will use that copy of the build item. If abuild can't find an item in the local forest, it will use the backing area to resolve that build item. Since abuild never attempts to build or otherwise modify an item in a backing area, backing areas must always be fully built on all platforms for which they will be used as backing areas. (For additional details on platforms, please see Chapter 5, *Target Types, Platform Types, and Platforms*, page 24.)

A build forest may declare multiple backing areas. To specify the location of your backing areas, create a file called *Abuild.backing* in the root directory of your build forest. As with the *Abuild.conf* file, the *Abuild.backing* file consists of colon-separated key/value pairs. The **backing-areas** key is followed by a space-separated list of paths to your backing areas. Backing area paths may be specified as either absolute or relative paths. The path you declare as a backing area may point anywhere into the forest that you wish to use as the backing area. It doesn't have to point to the root of the forest, and it doesn't have to point to a place in the forest that corresponds to the root of your forest.

When one forest declares another forest as a backing area, we say that the forest *backs to* its backing area. Creation and maintenance of backing areas is generally a function performed by the people who are in charge of maintaining the overall software baselines. Most developers will just set up their backing areas according to whatever instructions they are given. Having an external tool to create your *Abuild.backing* file is also reasonable. Note that *Abuild.backing* files should not generally be controlled in a version control system since they are a property of the developer's work area rather than of the software baseline. If they are controlled, they should generally not be visible outside of the developer's work area.

Note

Changing backing area configuration should generally be followed by a clean build. This is also true when a build item is removed from a local build tree and therefore causes the build item with that name to resolve to the copy in backing area. The reason is that changing the location of a build item changes the actual files on which the build target depends. If those dependencies are older than the last build time, even if they were newer than the files they replaced, make and ant will not notice because they use modification time-based dependencies. In other words, any operation that can replace one file with another file in such a way that the new file is not more recent than the last build should be followed by a clean build.

11.2. Resolving Build Items to Backing Areas

In this section, we will discuss backing areas from a functionality standpoint. This section presents a somewhat simplified view of how backing areas actually work, but it is good enough to cover the normal cases. To understand the exact mechanism that abuild uses to handle backing areas with enough detail to fully understand the subtleties of how they work, please see Section 33.3, “Traversal Details”, page 216.

The purpose of a backing area is to enable a developer to create a partially populated build tree and to fall back to a more complete area for build items that are omitted in the local build tree. A build forest may have any number of backing

areas, and backing areas may in turn have additional backing areas. There are a few restrictions, however. As with item and tree dependencies, there may be no cycles among backing area relationships. Additionally, if two unrelated backing areas supply items or trees with the same name, this creates an ambiguity, which abuild will consider an error. [1]

When you have one or more backing areas, any reference to a build item or build tree that is not found locally can be resolved in the backing area. What abuild essentially does is to maintain a list of available item and tree names, which it internally maps to locations in the file system. When you using a backing area, abuild uses the backing areas' lookup tables in addition to that from your own forest to resolve items and trees. [2] When a build item or tree is defined in a backing area and is also defined in your local forest, your local forest is said to *shadow* the item or tree. This is not an error. It is a normal case that happens when you are using backing areas. In most cases, your build forest will contain items that either exist now in the backing area or will exist there at some future point. This is because the backing area generally represents a more stable version of whatever project you are working on.

Note that since abuild refers to build items and trees by name and not by path, there are no restrictions about the location of build items in the local forest relative to where they appear in the backing area. This makes it possible for you to reorganize the build items or even the build trees in your local area without having to simultaneously change the backing area. There is only way in which use of backing areas affects how abuild resolves paths: if a directory named in a **child-dirs** key in some *Abuild.conf* does not exist and the forest has a backing area, abuild will ignore the non-existence of the child directory. (If you run with **--verbose**, it will mention that it is ignoring the directory, but otherwise, you won't be bothered with this detail.) This enables you to create sparsely populated build items without having to edit *Abuild.conf* files of the parents of the directories you have chosen to omit.

If this seems confusing, the best way to think about it is in terms of how this all interacts with a version control system. Typically, there is some master copy of the source code of a project in a version control system. There may be some stable trunk or branch in the version control system that is expected to be self-contained and operational. This is what would typically be checked out into a forest that would be fully built and used by others as a backing area. Then, individual developers would just check out the pieces of the system that they are working on, and set their backing area to point to the stable area. Since their checkouts would be sparse, there may be child directories that don't exist, but it wouldn't matter; once they check in their changes and the stable area from which the backing area is created gets updated, everything should be normal.

One side effect of this is that if you remove the directory containing a build item or tree from your local forest while using a backing area that still contains that item or tree, the thing you removed doesn't really go away from abuild's perspective. Instead, it just “moves” from the local build tree to the backing area. If it is actually your intention to *remove* the build item so that its name is not known to other build items in your build tree, you can do this by adding the name of the build item to the **deleted-items** key or the build tree to the **deleted-trees** key of your *Abuild.backing* file. This effectively blocks abuild from resolving items or trees with those names from the backing area. Most users will probably never use this feature and don't even need to know it exists, but it can be very useful under certain circumstances. When you tell abuild to ignore a tree in this way, it actually blocks abuild from seeing any items defined in the deleted tree. If you wanted to, you could create a new tree locally with the same name as the deleted tree, and the new tree and the old tree would be completely separate from each other. We present an example that illustrates the use of the **deleted-item** key in Section 11.5, “Deleted Build Item”, page 65.

11.3. Integrity Checks

In plain English, abuild guarantees that if ***A*** depends on ***B*** and ***B*** depends on ***C***, ***A*** and ***B*** see the same copy of ***C***. To be more precise, abuild checks to make sure that no build item in a backing area references as a dependency or plugin

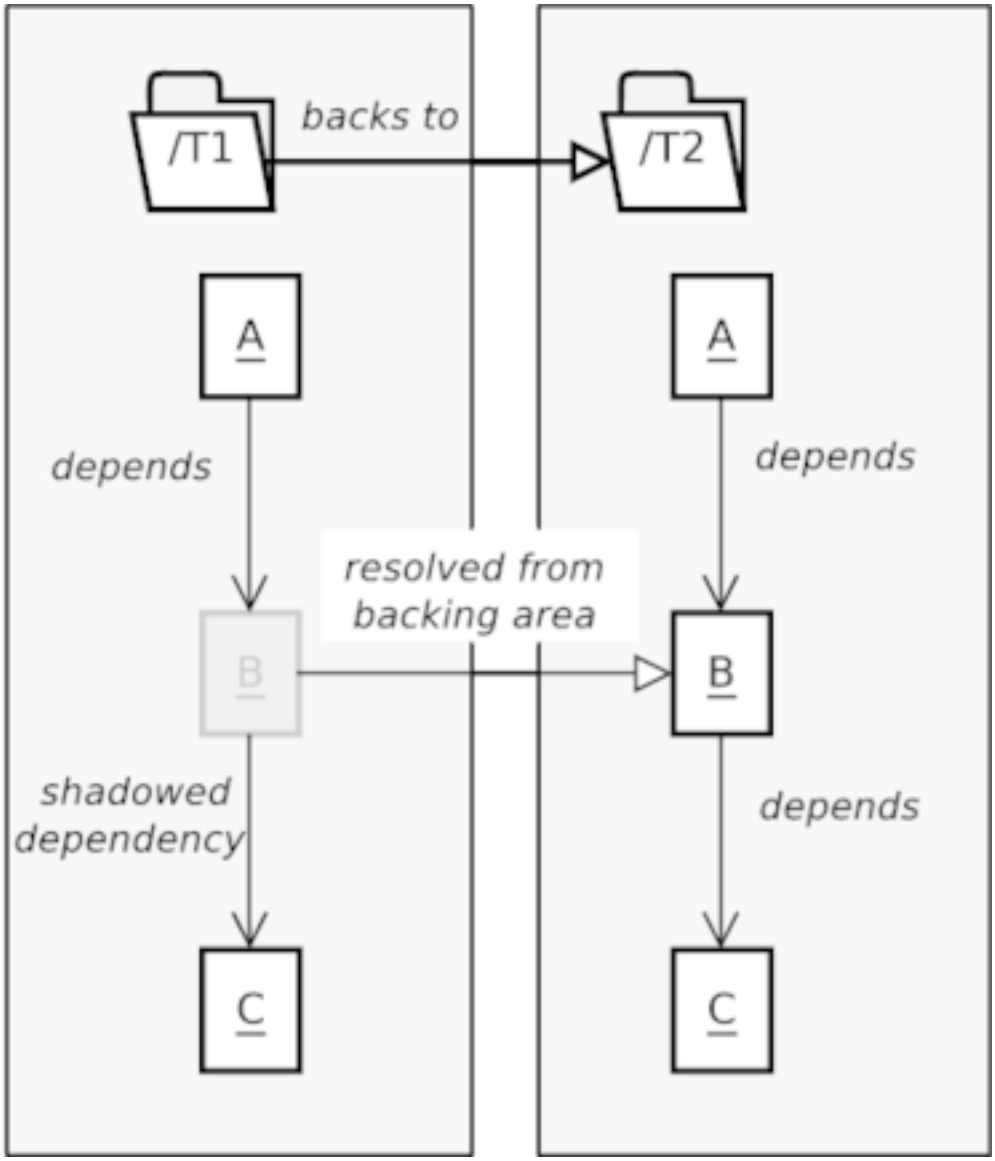
[1] What do we mean by “unrelated” backing areas? If your build forest declares *A* and *B* to be backing areas and *A* backs to *B*, abuild will notice this relationship and will ignore your mention of *B* as a backing area. You will still inherit items from *B*, but you will do so through *A* instead of getting them directly. Abuild doesn't consider this to be an error or even a warning since, for all you know, *A* and *B* may be independent, and *A* may be using *B* on a temporary or experimental basis. However, if you really want to know, abuild will tell you that it is ignoring *B* when you run it with the **--verbose** flag.

[2] The actual implementation differs from this description, but the effect is the same. For the real story, see Section 33.3, “Traversal Details”, page 216.

an item that is shadowed in the local forest. (Plugins are covered in Chapter 29, *Enhancing Abuild with Plugins*, page 185.)

We illustrate this in Figure 11.1, "Shadowed Dependency", page 61. Suppose that build items ***A***, ***B***, and ***C*** are defined in build tree *T2* and that ***A*** depends on ***B*** and ***B*** depends on ***C***. Now suppose you have a local build tree called *T1* that has *T2* as its backing area, and that you have build items ***A*** and ***C*** copied locally into *T1*, but that ***B*** is resolved in the backing area.

Figure 11.1. Shadowed Dependency



A in */T1* sees ***B*** in */T2* and ***C*** in */T1*, but ***B*** in */T2* sees ***C*** in */T2*. This means ***A*** in */T1* builds with two different copies of ***C***.

If you were to attempt to build ***A***, ***A*** would refer to files in ***B***, which comes from a backing area. ***B*** would therefore already be built, and it would have been built with the copy of ***C*** from the backing area. ***A***, on the other hand, would see ***C*** in the local build tree. That means that ***A*** is indirectly using two different copies of ***C***. Depending on what changes were made to ***C*** in the local build tree, this would likely cause the build of ***A*** to be unreproducible at best and

completely broken at worst. The situation of ***B*** coming from a backing area and depending on ***C***, which is shadowed locally, is what we mean when we say that ***B*** has shadowed dependencies. If you attempt to build in this situation, abuild will provide a detailed error message telling you which build items are shadowed and which other build items depend on them. One way to resolve this would be to copy the shadowed build items into your local build tree. In this case, that would mean copying ***B*** into *T1*. Another way to resolve it would be to remove ***C*** from your local area and allow that to be resolved in the backing area as well. This solution would obviously only be suitable if you were not working on ***C*** anymore.

11.4. Task Branch Example

In this example, we'll demonstrate a task branch. Suppose our task branch makes changes to *project* but not to *common* or *derived*. We can set up a new build forest in which to do our work. We would populate this build forest with whatever parts of *project* we wanted to modify. We have set up this forest in *doc/example/general/task*. Additionally, we have set this forest's backing area to *../reference* so that it would resolve any missing build items or trees to that location:

general/task/Abuild.backing

```
backing-areas: ../reference
```

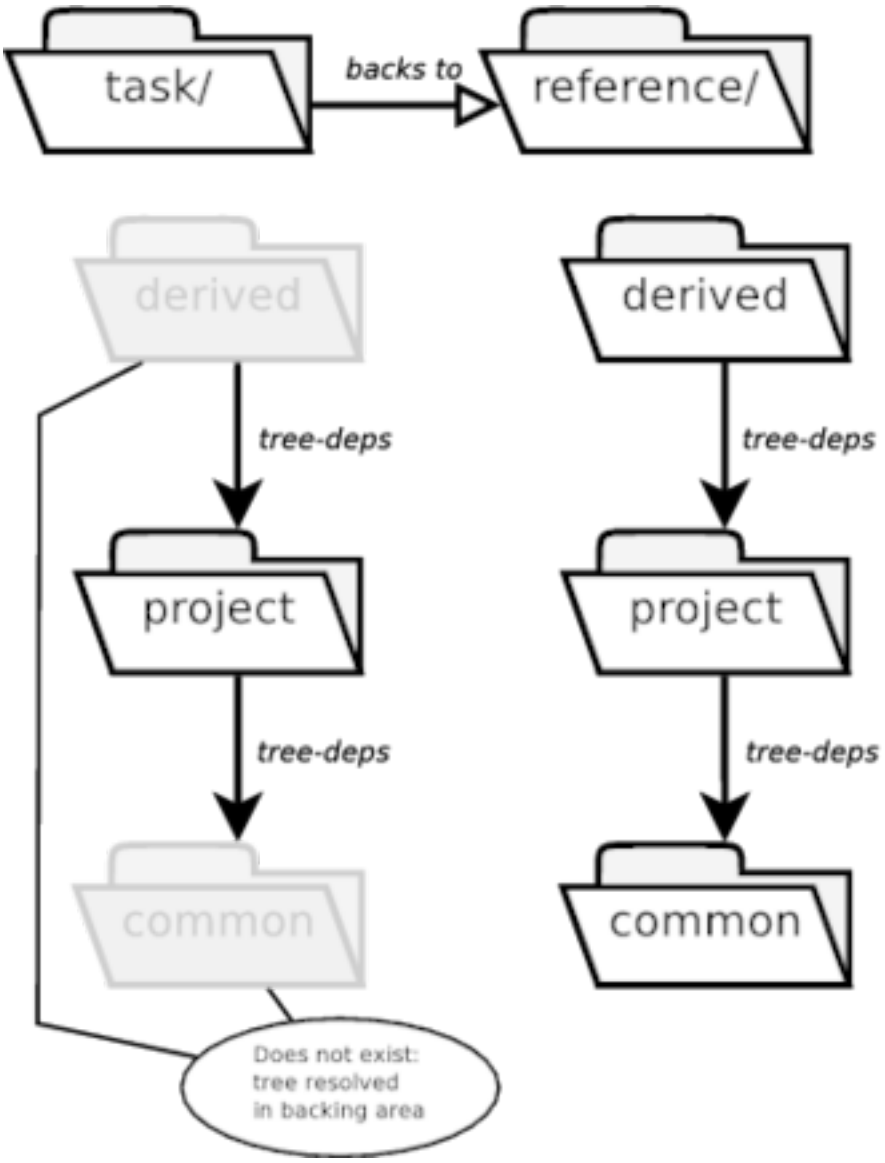
Note that, although we used a relative path for our backing area in this example, we would ordinarily set our backing area to an absolute path. We use a relative path here only so that the examples can remain independent of the location of *doc/example*. Since we are not making modifications to any build items in *common* or *derived*, we don't have to include those build trees in our task branch. Note that our forest root *Abuild.conf* still lists *common* and *derived* as children, since it is just a copy of the root *Abuild.conf* from reference:

general/task/Abuild.conf

```
child-dirs: common project derived
```

Since this forest has a backing area, abuild ignores the fact that the *common* and *derived* directories do not exist. For a diagram of the task branch build trees, see Figure 11.2, “Build Trees in *general/task*”, page 63.

Figure 11.2. Build Trees in *general/task*



The *derived* build tree declares a tree dependency on the *project* build tree. The *project* build tree declares a tree dependency on the *common* build tree. Since the *common* and *derived* build trees are not shadowed in the *task* branch, those trees are resolved in the backing area, *reference*, instead.

As always, for this example to work properly, our backing area must be fully built. If you are following along, to make sure this is the case, you should run **abuild --build=all** in *reference/derived*. Next run **abuild --build=deptrees no-op** in *task/project*. This generates the following output:

task-project-no-op.out

```
abuild: build starting
abuild: project-lib.src (abuild-<native>): no-op
abuild: project-lib.test (abuild-<native>): no-op
abuild: project-main.src (abuild-<native>): no-op
abuild: build complete
```

This includes only items in our task branch. No items in our backing area are included because abuild never attempts to build or modify build items in backing areas.

If you study *include/ProjectLib.hpp* and *src/ProjectLib.cpp* in *task/project/lib* in comparison to their counterparts in *reference/project/lib*, you'll notice that the only change we made in this task branch is the addition of an optional parameter to ***ProjectLib***'s constructor. We also updated the test suite to pass a different argument to ***ProjectLib***. This new value comes from a new build item we added: ***project-lib.extra***. To add the new build item, we created *task/project/lib/extra/Abuild.conf*: and also added the *extra* directory in *task/project/lib/Abuild.conf*:

general/task/project/lib/extra/Abuild.conf

```
name: project-lib.extra
platform-types: native
```

general/task/project/lib/Abuild.conf

```
name: project-lib
child-dirs: src test extra
deps: project-lib.src
```

We didn't modify anything under *task/project/main* at all, but we included it in our task branch so we could run its test suite. Remember that abuild won't try to build the copy of ***project-main*** there, and even if it did, that copy of ***project-main*** would not see our local copy of ***project-lib***: it would see the copy in its own local build tree, which we have shadowed. This is an example of a shadowed dependency as described in Section 11.3, “Integrity Checks”, page 60. This is the output we see when running **abuild --build=deptrees check** from *task/project*:

task-project-check.out

```
abuild: build starting
abuild: project-lib.src (abuild-<native>): check
make: Entering directory `--topdir--/general/task/project/lib/src/abuild\
\-<native>'
Compiling ../ProjectLib.cpp as C++
Creating project-lib library
make: Leaving directory `--topdir--/general/task/project/lib/src/abuild-\
\<native>'
abuild: project-lib.test (abuild-<native>): check
make: Entering directory `--topdir--/general/task/project/lib/test/abuil\
\d-<native>'
Compiling ../main.cpp as C++
Creating lib_test executable

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/lib.test
lib  1 (test lib class)                                    ... PASSED

Overall test suite                                         ... PASSED

TESTS COMPLETE.  Summary:
```

```
Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/task/project/lib/test/abuild\
\-<native>'
abuild: project-main.src (abuild-<native>): check
make: Entering directory `--topdir--/general/task/project/main/src/abuil\
\d-<native>'
Compiling ../main.cpp as C++
Creating main executable

********************************
STARTING TESTS on ---timestamp---
********************************


Running ../qtest/main.test
main  1 (testing project-main)                                  ... PASSED

Overall test suite                                              ... PASSED

TESTS COMPLETE.  Summary:

Total tests: 1
Passes: 1
Failures: 0
Unexpected Passes: 0
Expected Failures: 0
Missing Tests: 0
Extra Tests: 0

make: Leaving directory `--topdir--/general/task/project/main/src/abuild\
\-<native>'
abuild: build complete
```

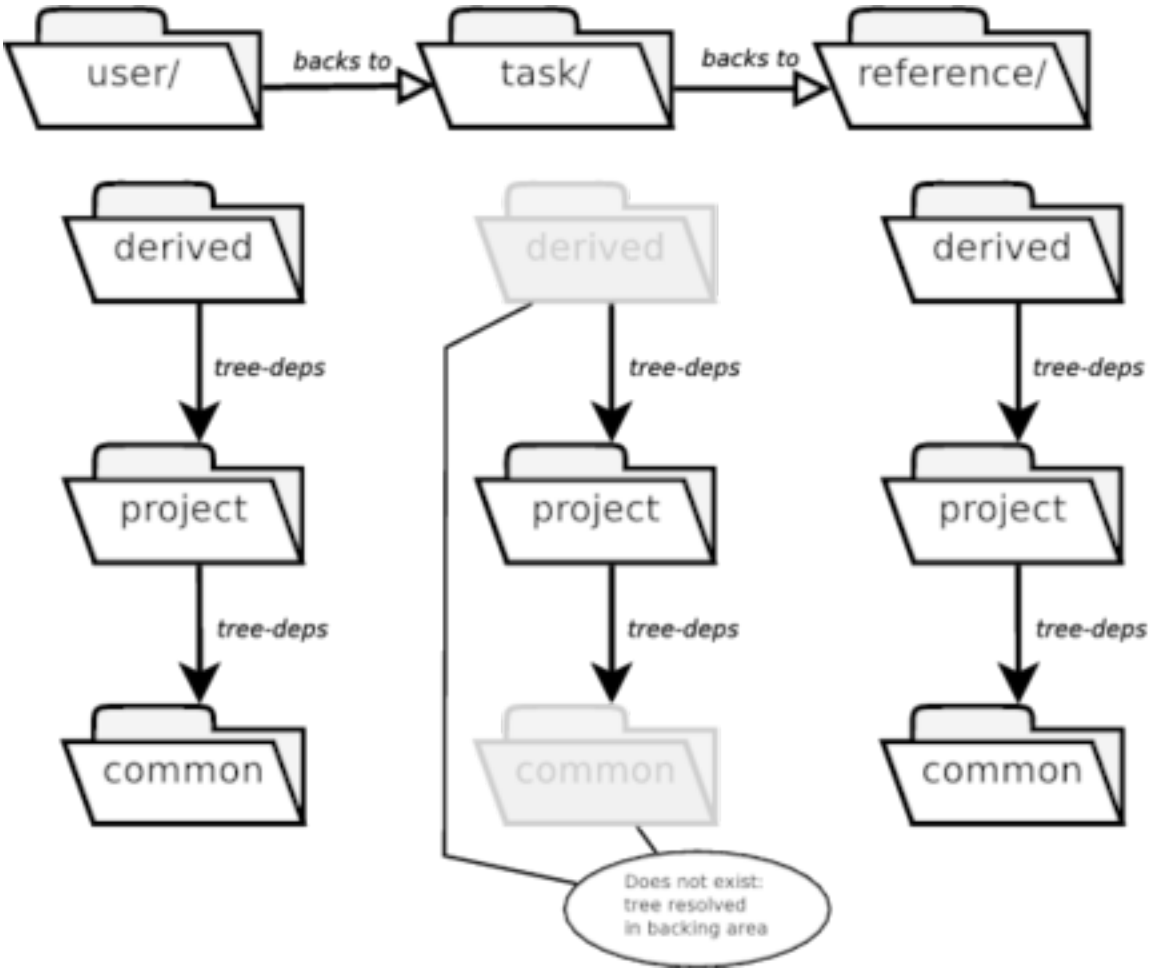
As with the **no-op** build, we only see output relating to local build items, not to build items in our backing areas as they are assumed to be already built.

11.5. Deleted Build Item

Here we present a new forest located under *doc/example/general/user*. This forest backs to the *task* forest from the previous example. We will use this forest to illustrate the use of the **deleted-item** key in the *Abuild.backing*.

Suppose we have a user who is working on changes that are related in some way to the task branch. We want to create a user branch that backs to the task branch. Our user branch contains all three trees: *common*, *project*, and *derived*. We will ignore *derived* for the moment and focus only *common* and *project*. For a diagram of the user build trees, see Figure 11.3, “Build Trees in *general/user*”, page 66.

Figure 11.3. Build Trees in *general/user*



The *user* forest backs to the *task* forest. All trees are present, so they all resolve locally.

Observe that *common* contains only the *lib1* directory and that *project* contains only the *lib* directory. We make a gratuitous change to a source file in ***common-lib1.src*** just as another example of shadowing a build item from our backing area.

In *project*, we have made changes to *project-lib* to make use of private interfaces, which we discuss in Chapter 23, *Interface Flags*, page 150 and will ignore for the moment. We have also deleted the new build item ***project-lib.extra*** that we added in the task branch. To delete the build item, we removed the *extra* directory from *project/lib* and from the **child-dirs** key in *project/lib/Abuild.conf*:

general/user/project/lib/Abuild.conf

```
name: project-lib
child-dirs: src test
deps: project-lib.src
```

That in itself was not sufficient since, even though the *extra* directory is no longer present in the **child-dirs** key of ***project-lib***'s *Abuild.conf*, we would just inherit ***project-lib.extra*** from our backing area. To really delete the build item, we also had to add a **deleted-item** key in *user/Abuild.backing*:

general/user/Abuild.backing

```
backing-areas: ../task
deleted-items: project-lib.extra
```

This has effectively prevented abuild from looking for ***project-lib.extra*** in the backing area. If any build item in the local tree references ***project-lib.extra***, an error will be reported because abuild now considers that to be an unknown build item.

Although we don't present any examples of using **deleted-tree**, it works in a very similar fashion. Ordinarily, any build tree you do not have locally will be inherited from the backing area. If your intention is to change the code so that it no longer uses a particular tree, and you want to make sure that that tree is not available at all in your local area, you can delete it using **deleted-tree**. However, if you simply remove it from all your **tree-deps** directives, there is no risk of your using its items by accident. As such, most people will probably never need to use the **deleted-tree** feature.

Chapter 12. Explicit Read-Only and Read/Write Paths

One of the significant defining features of abuild is that it will automatically build items to satisfy dependencies. Most of the time, this is useful and helpful behavior, but there are certain cases in which it can actually get in the way. For example, you may have one build tree that provides common code, which you may want to build manually in advance of building everything else. Then you may want to kick off parallel builds of separate dependent trees on multiple platforms simultaneously and be able to be sure that they won't step on each other by all trying to build the shared tree at the same time. In cases like this, you might want to tell abuild to assume the shared tree is built and to treat it as read-only.

To support this and similar scenarios, abuild allows you to explicitly designate specific paths as read-only on the command line. [1] Most of the time, specifying a read-only path is as simple as invoking abuild with the **--ro-path=*`directory`*** option for the directory you want to treat as read-only. There may cases, however, where you want a much more specific degree of control. When you need it, it's there. Here we describe the exact behavior of the **--ro-path** and **--rw-path** options.

- Both **--ro-path=*`dir`*** and **--rw-path=*`dir`*** may appear multiple times.
- If neither option appears, all build items are writable. (Except those in backing areas; backing areas are always read-only.)
- If only **--ro-path** appears, build items are writable by default, and only those located under any specified read-only path are read-only.
- If only **--rw-path** appears, build items are *read-only* by default, and only build items located under a directory specified with **--rw-path** are writable.
- If both **--ro-path** and **--rw-path** are specified:
  - Either every **--ro-path** must be a path under some **--rw-path**, in which case build items are read-only by default,
  - or every **--rw-path** must be path under some **--ro-path**, in which case build items are writable by default.

  In this case, the writability of items is determined by the lowest directory actually specified (start with the item's directory and walk up the file system until you find a directory explicitly mentioned), using the default of none is found.

This seems more complicated than it really is, so let's look at a simple example. Suppose you have the directory structure *a/b/c/d*. If you specified **--ro-path=a/b --rw-path=a/b/c**, all read/write paths are under some read only path, so build items are writable by default. Everything under *a/b/c* is writable, and everything under *a/b* that is *not* under *a/b/c* is read-only. Use of **--ro-path** and **--rw-path** together basically lets you make a particular area read only and then give exceptions. Likewise, you can make everything read-only by default, and then make only a specific directory read-write, again make exceptions to that.

These rules make it possible to unambiguously create any combination of read-only/writable build items without having the order of the arguments matter. If you're sufficiently determined, you can use this mechanism to precisely control which items should be read-only and which should be writable.

[1] In abuild 1.0, we had a different mechanism for addressing this need: read-only externals. There were several problems with read-only externals, though: they were ambiguous since whether a tree was read-only or not would depend on how abuild came to know about it through other trees, they were not granular as you could only control this at the tree level, and they were inflexible: you might set them up to address the needs of a particular build, and then have them get in the way of other builds. When externals were replaced by named trees and tree dependencies, we dropped support for read-only externals and replaced them with read-only paths, which are much more flexible and which make the decision a function of the specific build rather than of the build trees, as it always should have been.

Paths specified may be absolute or relative. Relative paths are resolved relative to the starting directory of abuild. They are converted internally to absolute paths after any **-C *start-directory*** option is evaluated.

Chapter 13. Command-Line Reference

This chapter presents full detail about how to invoke abuild from the command line. Some of functionality described here is explained in the chapters of Part III, “Advanced Functionality”, page 78.

13.1. Basic Invocation

When running abuild, the basic invocation syntax is as follows:

```
abuild [options] [definitions] [targets]
```

Options, definitions, and targets may appear in any order. Any argument that starts with a dash (“-”) is treated as an option. Any option of the form **VAR=value** is considered to be a definition. Anything else is considered to be a target.

13.2. Variable Definitions

Arguments of the form **VAR=value** are variable or parameter definitions. Variables defined in this way are made available to all backends. These can be used to override the value of interface variables or variables set in backend build files.

For the make backend, these variable definitions are just passed along to make.

For the Groovy backend, these variables are stored in a manner such that *abuild.resolve* will give them precedence over normal parameters or interface variables. They are also defined as properties in the ant project.

For the deprecated xml-based ant framework, these definitions are made available as ant properties that are defined prior to reading any generated or user-provided files.

13.3. Informational Options

These options print information and exit without building anything.

--dump-build-graph
: Dump to standard output the complete build graph consisting of items and platforms. This is primarily useful for debugging abuild or diagnosing unusual problems relating to which items are built in which order. The build graph output data conforms to a DTD which can be found in *doc/build_graph.dtd* in the abuild distribution. The contents of the DTD can also be found in Appendix H, ***--dump-build-graph** Format*, page 305. Although nothing is built when this option is specified, abuild still performs complete validation including reading of all the interface files. The build graph is discussed in Section 33.6, “Construction of the Build Graph”, page 218. For additional ways to use the build graph output, see also Chapter 32, *Sample XSL-T Scripts*, page 214.

--dump-data
: Dump to standard output all information computed by abuild. Useful for debugging or for tools that need in-depth information about what abuild knows. **--dump-data** is mutually exclusive with running any targets. If you need to see **--dump-data** output and build targets at the same time, use **--monitored** instead (see Chapter 31, *Monitored Mode*, page 212). For details about the format generated by **--dump-data**, please see Appendix F, ***--dump-data** Format*, page 296. For additional ways to use the build graph output, see also Chapter 32, *Sample XSL-T Scripts*, page 214.

--find item-name
: Print the name of the tree that contains item *item-name*, and also print its location.

--find=tree:tree-name
Print the location of the root build item of build tree *`tree-name`*.

--help|-H
Print a brief introduction to abuild's help system. For additional details, see Chapter 8, *Help System*, page 37. For the text of all help files that are provided with abuild, see Appendix E, *Online Help Files*, page 270.

--list-platforms
Print the names of all object-code platforms categorized by platform type and build tree, and indicate which ones would be built by default. Note that abuild may build on additional platforms beyond those selected by default in order to satisfy dependencies from other items.

--list-traits
Print the names of all traits known in the local build tree, its tree dependencies, and its backing areas. This is the list of traits that are available for use on the command line with the **--only-with-traits** and **--related-by-traits** options.

--print-abuild-top
Print the path to the top of abuild's installation.

-V|--version
Print the version number of abuild.

13.4. Control Options

These options change some aspect of how abuild starts or runs.

-C start-directory
Change directories to the given directory before building.

--clean-platforms=pattern
Specify a pattern that restricts which platform directories are removed by any abuild clean operation. This argument may be repeated any number of times. The *`pattern`* given can be any valid shell-style wild-card expression. Any output directory belonging to any pattern that matches any of the given clean patterns will be removed. All other output directories will be left alone. This can be useful for removing only output directories for platforms we no longer care about or for other selective cleanup operations.

--compat-level=version
Set abuild's compatibility level to the specified version, which may be either `1.0` or `1.1`. You may also place the compatibility level version in the *ABUILD_COMPAT_LEVEL* environment variable. By default, early prerelease versions of abuild attempt to detect deprecated constructs from older versions and issue warnings about their use, while final versions operate with deprecation support disabled by default. Setting the compatibility level to a given version causes abuild to not recognize constructs deprecated by that version at all. For example, in compatibility level `1.1`, use of the **this** key in *Abuild.conf* would result in an error about an unknown key rather than being treated as if it were **name**, and the make variable *BUILD_ITEM_RULES* would be treated like any ordinary variable and would not influence the build in any way. See also **--deprecation-is-error**.

--deprecation-is-error
Ordinarily, abuild detects deprecated constructs, issues warnings about them, and continues operating by mapping deprecated constructs into their intended replacements. When this option is specified, any use of deprecated constructs are detected and reported as errors instead of warnings. Note that this is subtly different from specifying **--compat-level** with the current major and minor versions of abuild. For example, if **--deprecation-is-error** is specified, use of the make variable *BUILD_ITEM_RULES* will result in an error message, while if **--compat-level=1.1** is specified, the variable will simply be ignored. A good upgrade strategy is to use **--deprecation-is-error** to first test to make sure you've successfully eliminated all deprecated constructs, and then to use **--compat-lev-**

el (or to set the *ABUILD_COMPAT_LEVEL* environment variable) to turn off abuild's backward compatibility support, if desired.

-e | --emacs
: Tell ant to run in emacs mode by passing the **-e** flag to it and also setting the property *abuild.private.emacs-mode*. Ant targets can use this information to pass to programs whose output may need to be dependent upon whether or not emacs mode is in effect.

--find-conf
: Locates the first directory at or above the current directory that contains an *Abuild.conf* file, and changes directories to that location before building.

--full-integrity
: Performs abuild's integrity checks for all items in the local tree, tree dependencies, and backing areas. Ordinarily, abuild performs its integrity check only for items that are being built in the current build. The **--full-integrity** flag would generally be useful only for people who are maintaining backing areas that are used by other people. For detailed information about abuild's integrity checks, please see Section 11.3, “Integrity Checks”, page 60.

-jn| --jobs=n
: Build up to *n* build items in parallel by invoking up to *n* simultaneous instances of the backend. Does not cause the backend to run multiple jobs in parallel. See also **--make-jobs**.

--jvm-append-args ... --end-jvm-args
: Appends any arguments between **--jvm-append-args** and **--end-jvm-args** to the list of extra arguments that abuild passes to the JVM when it invokes the java builder backend. This option is intended for use in debugging abuild. If you have to use it to make your build work, please report this as a bug.

--jvm-replace-args ... --end-jvm-args
: Replaces abuild's internal list of extra JVM arguments with any arguments between **--jvm-replace-args** and **--end-jvm-args**. This option is intended for use in debugging abuild. If you have to use it to make your build work, please report this as a bug.

-k | --keep-going
: Don't stop the build after the first failed build item, but instead continue building additional build items that don't depend on any failed items. Also tells backend to continue after its first failure. Even with **-k**, abuild will never try to build an item if any of its dependencies failed. This behavior may be changed by also specifying **--no-dep-failures**.

--make
: Terminate argument parsing and pass all remaining arguments to make. Intended primarily for debugging.

--make-jobs[=n]
: Allow make to run up to *n* jobs in parallel. Omit *n* to allow make to run as many jobs as it wants. Be aware that if this option is used in combination with **--jobs**, the total number of threads could potentially be the product of the two numerical arguments.

  Note that certain types of make rules and certain may cause problems for parallel builds. For example, if your build involves invoking a compiler or other tool that writes poorly named temporary files, it's possible that two simultaneous invocations of that tool may interfere with each other. Starting with abuild 1.1, it is possible to place **attributes: serial** in a make-based build item's *Abuild.conf* file to prevent **--make-jobs** from applying to that specific item. This will force serial compilation of items that you know don't build properly in parallel. This can be useful for build items that use the *autoconf* rules, which are known to sometimes cause trouble for parallel builds.

--monitored
: Run in monitored mode. For details, see Chapter 31, *Monitored Mode*, page 212.

-n
: Have the backend print what it would do without actually doing it.

--no-dep-failures
: Must be combined with **-k**. By default, abuild does not attempt to build any items whose dependencies have failed even if **-k** is specified. When the **--no-dep-failures** option is specified along with **-k**, abuild will attempt to build items even if one or more of their dependencies have failed. Using **-k** and **--no-dep-failures** together enables abuild to attempt to build everything that the backends will allow. Note that cascading errors (*i.e.*, errors resulting from earlier errors) are likely when this option is used.

--platform-selector=selector | -p selector
: Specify a platform selector. This argument may be repeated any number of times. Later instances supersede earlier ones when they specify selection criteria for the same platform type. When two selectors refer to different platform types, both selectors are used. Platform selectors may also be given in the *ABUILD_PLATFORM_SELECTORS* environment variable. For details on platform selectors, see Section 24.1, “Platform Selection”, page 155.

--ro-path=path
: Indicate that *`path`* is to be treated as read-only by abuild during build or clean operations. For details on using explicitly read-only and read/write paths, see Chapter 12, *Explicit Read-Only and Read/Write Paths*, page 68.

--rw-path=path
: Indicate that *`path`* is to be treated as read-write by abuild during build or clean operations. For details on using explicitly read-only and read/write paths, see Chapter 12, *Explicit Read-Only and Read/Write Paths*, page 68.

13.5. Output Options

These options change the type of output that abuild generates.

--buffered-output
: Cause abuild to buffer the output produced by each individual item's build and display it contiguously after that build completes. For additional details, see Chapter 20, *Controlling and Processing Abuild's Output*, page 119.

--error-prefix=prefix
: Prepend the given prefix string to every error message generated by abuild and to every line written to standard error by any program abuild invokes. See also **--output-prefix**. For additional details, see Chapter 20, *Controlling and Processing Abuild's Output*, page 119.

--interleaved-output
: In a multithreaded build, cause abuild to prepend each line of output (normal or error) with an indicator of the build item that was responsible for producing it. Starting in abuild version 1.1.3, this is the default for multithreaded builds. For additional details, see Chapter 20, *Controlling and Processing Abuild's Output*, page 119.

--output-prefix=prefix
: Prepend the given prefix string to every line of non-error output generated by abuild and to every line written to standard output by any program abuild invokes. See also **--error-prefix**. For additional details, see Chapter 20, *Controlling and Processing Abuild's Output*, page 119.

--raw-output
: Prevent abuild from doing any kind of capture or filtering of the output produced by any item's build. This option also makes abuild's standard input available to any program that abuild invokes. This is the default for single-threaded builds and was the behavior for all builds prior to abuild version 1.1.3. For additional details, see Chapter 20, *Controlling and Processing Abuild's Output*, page 119.

--silent
: Suppress most non-error output. Also tells the backend build tools to generate less output.

--verbose
: Generate more verbose output. Also tells the backend build tools to generate more output.

13.6. Build Options

These options tell abuild what to build and what targets to apply to items being built.

--apply-targets-to-deps
: Ordinarily, any explicitly specified targets are applied only to items that were directly selected for inclusion in the build set. With this flag, they are applied to all items being built, including recursively expanded dependencies. When used with a clean set, this option causes the clean set to expanded to include dependencies, which is otherwise not done. For detailed information about target selection, please see Chapter 9, *Telling Abuild What to Build*, page 38.

--build=set | -b set
: Specify which build items should be built. The default is to use the build set **current**, which builds the current item and all of its dependencies. For additional details including a list of valid values for `set`, see Chapter 9, *Telling Abuild What to Build*, page 38.

--clean=set | -c set
: Run **abuild clean** in all items in the build set. The same build sets are defined as with the **--build** option. Unlike build sets, clean sets are not expanded to include dependencies (unless **--apply-targets-to-deps** is specified), and build items are not cleaned in dependency order. No targets may be specified in conjunction with this option. For additional details including a list of valid values for `set`, see Chapter 9, *Telling Abuild What to Build*, page 38. See also the description of the **--clean-platforms** (in Section 13.4, “Control Options”, page 71) to learn about restricting which platform directories are removed.

--dump-interfaces
: Cause abuild to create interface dump files in the output directories of every writable build item, including those that don't build anything. This option can be useful for tracking down problems with interface variables. For more information, see Section 17.6, “Debugging Interface Issues”, page 94.

--no-deps
: Prevent abuild from attempting to build any dependencies of the current build item before building the item itself. The **--no-deps** option may not be combined with a build set.

--only-with-traits=trait[,trait,...]
: Exclude from the initial build set any items that do not contain all of the named traits. As always, all dependencies of any item in the reduced build set will remain in the build set regardless of what traits they have. If not accompanied by the **--related-by-traits** option, any explicitly named targets will be applied only to items that have all of the named traits. Other items (those they depend on) will be built with the default **all** target. If accompanied by the **--related-by-traits** option, the **--related-by-traits** option's behavior with respect to explicit targets takes precedence. For more information about traits, see Section 9.5, “Traits”, page 42.

--related-by-traits=trait[,trait,...]
: Expand the build set with items that have all of the named traits relative to any item already in the build set. Specifying this option also causes any explicitly specified targets to be run only for those items. The default target **all** is run for all other build items in the build set. For more information about traits, see Section 9.5, “Traits”, page 42. When combined with **--repeat-expansion**, this process is repeated until no more items are added.

--repeat-expansion
: Instruct abuild to apply build set expansion based on traits (**--related-by-traits**) or or reverse dependencies (**--with-rdeps**) repeatedly after adding dependencies of newly added items until no further expansion of the build set results.

--with-rdeps
: Expand the build set by adding all reverse dependencies of any item already in the build set. As always, any additional dependencies of newly added items are also added. When specified with **--repeat-expansion**, addition of reverse dependencies is repeated (after adding additional dependencies) until no further expansion of the build set results.

13.7. General Targets

Abuild's backends define several targets that are available for use from the command line, so you can rely on these targets being defined. [1]

all
: This is the default target. It is used to build all products that are intended for use by the end user or by other build items.

check
: This target ensures that the local build item is built and then runs its automated test suite, if any. For this to do anything, the build item must have a test suite implemented with a test framework that is integrated with abuild or that is made available with a plugin. Abuild is integrated with QTest and, for Java-based build items, also with JUnit. The **check** target is not automatically run by the default target; it must be requested specifically.

clean
: This target removes any output directories that abuild thinks it created. (Output directories are discussed in Section 5.3, “Output Directories”, page 26.) Well-behaved abuild rules, including all the rules that are a standard part of abuild, won't create any files or directories outside of these locations. See also the description of the **--clean-platforms** (in Section 13.4, “Control Options”, page 71) to learn about restricting which platform directories are removed.

doc
: This target is provided for building documentation that is extracted from source code. The **doc** target is not automatically run by the default target; it must be requested explicitly. It depends on the **all** target. There is no internal support for document generation in the make backend, so this capability must be provided by a plugin. For Groovy/ant builds, there is built-in support for javadoc, but it is minimal and will likely have to be supplemented for any major documentation effort. A contributed plugins to support doxygen is available in *abuild-contrib*, which is released separately from abuild.

no-op
: This target does nothing other than printing the name and platform of each build item in the build set, but using it still causes abuild to perform all the same validations it would perform if it were going to build something. The **no-op** target can be used to get a complete list of all the items and platforms that would be built if building a given build set and will also verify that there are no errors in any *Abuild.conf* files. Note that *Abuild.interface* files are not read when invoking the **no-op** target.

test
: This target is a synonym for **check**.

test-only
: This target runs any automated test suites but does not first try to build. In other words, the **test-only** target does not depend on the **all** target like the **check** and **test** targets do. This can be useful for running a test suite on a build item without first rebuilding it or for running all the test suites on a build tree that you know is up to date because you just built it.

[1] When the *Abuild-ant.xml* build file is used with the deprecated xml-based ant backend, it is up to the author of the build file to provide these targets, and all bets are off.

Chapter 14. Survey of Additional Capabilities

By now, you should have a pretty good feel for what abuild can do and how to use it in several situations. The remaining chapters of this document cover advanced topics and present examples for solving a wide variety of problems. Although later chapters sometimes build on information presented in earlier chapters, many of the remaining chapters and examples can probably be understood on their own. It should therefore be safe to focus your attention on the material that is of interest or potential use to you.

Part III, "Advanced Functionality", page 78 opens with detailed descriptions of abuild's configuration files and interface subsystem. It then continues with explorations of several specific problems. We present here a brief list of problems that are addressed in the remaining chapters:

Controlling and Processing Abuild's Output
: Abuild's output is primary intended to be useful to human readers, but there are a number of capabilities (introduced in version 1.1.3) that can make it easier to programmatically parse abuild's output or to help make it easier to look at the output of a large build. In Chapter 20, *Controlling and Processing Abuild's Output*, page 119, we discuss ways to distinguish normal output from error messages and ways to clearly associate each line of abuild's output with the build item whose build produced it.

Shared Libraries
: Abuild includes support for creating shared libraries on UNIX platforms and DLLs on Windows platforms. In Chapter 21, *Shared Libraries*, page 123, we describe the process and explore some of the other concerns you have to consider when using shared libraries with abuild.

Build Item Rules and Code Generators
: Abuild allows build items to supply custom rules, most often for supporting automatic code generation. In Chapter 22, *Build Item Rules and Automatically Generated Code*, page 129, we discuss code generators for make-based and Groovy-based builds.

Private Interfaces
: In general, abuild is designed such that all build item interfaces automatically inherit through the dependency chain. There are some cases when it may be desirable for a build item to have an expanded interface that is available to certain build items upon request. In Chapter 23, *Interface Flags*, page 150, we introduce a feature of abuild designed to solve this problem and present an example of using it to implement private interfaces.

Cross-Platform Development
: Abuild's platform system is designed to make building on multiple platforms as easy as possible. If a build item can be built on multiple platforms, abuild will generally sort out all the details of which build of one item another item should depend on. There are times, however, when it is necessary to take control over this behavior. We discuss this problem in Chapter 24, *Cross-Platform Support*, page 155.

Mixed Classification Development
: We all know that security is increasingly important in the software community. In some cases, it may be necessary to create collections of software that are only allowed to run or even exist in secure environments. In Chapter 25, *Build Item Visibility*, page 166, we describe how to use abuild's build item visibility feature along with tree dependencies to create a mixed classification development environment, and we present an example that illustrates one implementation strategy.

Whole Library Support
: Ordinarily, when an application links with a library, only functions that are actually called are linked into the executable. On platforms that support this, abuild allows you to specify that the entire contents of a library archive

should be included in an executable. In Chapter 26, *Linking With Whole Libraries*, page 176, we describe why you might want to do this and how to do it.

Opaque Wrappers
: Some development problems require one interface to be created that opaquely hides another interface. Since abuild's default behavior is to make all interfaces inherit through the dependency chain, special constructs are required to implement opaque wrappers. In Chapter 27, *Opaque Wrappers*, page 179, we present the mechanisms required to make this work.

Optional Dependencies
: The goal of loose integration between software components can often be best served by allowing different components to make themselves known to the system at runtime. However, there are instances in which a tighter, compile-time integration may be required with optional components. In Chapter 28, *Optional Dependencies*, page 181, will illustrate how abuild allows you to declare tree and item dependencies as optional and then create code that is conditional upon whether the optional dependency is satisfied.

Plugins
: There are certain tasks that go beyond simply building targets and making them available. Examples include adding support for new compilers and performing extra validations that go beyond what can be easily expressed using abuild's built-in mechanisms. In Chapter 29, *Enhancing Abuild with Plugins*, page 185, we present a plugin framework that can be used to extend abuild in certain ways.

In addition to the above topics, we explore some details of how abuild works behind the scenes and present guidelines for how to use abuild in the safest and most effective way. The table of contents at the beginning of Part III, "Advanced Functionality", page 78 includes a complete list of chapters, and each chapter starts with some introductory text that describes the material it covers.

Part III. Advanced Functionality

In this part of the manual, we cover the remaining information about abuild's features in detail. This part contains complete reference guides to abuild's configuration files, discussions of more advanced topics, and numerous examples to illustrate how to solve specific build problems with abuild. By the end of this part, you should be able to use abuild for a wide range of build problems.

Chapter 15. The *Abuild.conf* File

The *Abuild.conf* file is the fundamental configuration file that describes each build item and the relationships between build items. It contains information about dependencies, file system locations, and platform support. It explicitly does not contain any information about how to build a particular build item or what targets are built.

15.1. *Abuild.conf* Syntax

Every build item must contain *Abuild.conf*. The *Abuild.conf* file is a simple text file consisting of colon-separated key/value pairs. Blank lines and lines that start with # are ignored. Long lines may be continued to the next line by ending them with a backslash character (\). Certain keys are permitted for some kinds of build items and not for others. For a discussion of different types of build items, please see Section 4.5, “Special Types of Build Items”, page 21.

The following keys are supported in *Abuild.conf*:

attributes
: This is a “catch-all” key whose value is a list of white-space separate keywords that assign certain specific attributes to a build item. The following attributes are supported:

- **serial**: valid only for build items that are built using the make backend, where it prevents the **--make-jobs** option from applying to that build item, effectively forcing it to build serially

build-also
: This key contains a list of whitespace-separated build items. Whenever abuild adds a given item to a build set, it also adds any items listed in its **build-also** key to the build set. No dependency relationship or any other relationship is implied. This is useful for creating pseudo-top-level build items that serve as starting points for multiple builds.

child-dirs
: This key is used to specify all subdirectories of this item that contain additional *Abuild.conf* files. The value is a whitespace-separated list of relative paths, each of which must point down in the file system.

  A child directory may be followed by the **-optional** flag, in which case abuild will not complain if the directory doesn't exist. This can be especially useful for high-level *Abuild.conf* files whose children may correspond to optional dependencies, optional build trees, or self-contained trees that may or may not be included in a particular configuration.

  If a child directory contains more than one path element, the intermediate directories may not contain their own *Abuild.conf* files. (In other words, you can't skip over a directory that has an *Abuild.conf* file in it.)

deps
: This key's value is a whitespace-separated list of the names of build items on which this build item depends. This is the sole mechanism within abuild to specify inter-build-item dependencies. Any dependency in this list may be optionally followed by one or more **-flag=*`interface-flag`*** arguments. This causes the *`interface-flag`* interface flag to be set when this build item reads the interface of the dependency (see Chapter 23, *Interface Flags*, page 150). It is also possible to specify a **-platform=*`selector`*** option to a dependency to specify which of the dependency's platforms applies to this dependency (see Section 24.3, “Explicit Cross-Platform Dependencies”, page 158). Dependencies may be specified as optional by following the dependency name with the **-optional** flag (see Chapter 28, *Optional Dependencies*, page 181).

description
: This key can be used to add an information description to the build item. Description information is intended to be human readable. If present, it will be included in the output to **abuild --dump-data**. Providing a description here

rather than just by using a comment in the *Abuild.conf* file can be useful to other programs that provide additional visualization of build items. For adding information that you may wish to categorize items for build purposes, use traits instead (see Section 9.5, “Traits”, page 42). The description field is only permitted for named build items, though comments may appear in any *Abuild.conf* file.

name
This key is used to set the name of the build item. Build item names consist of period-delimited segments. Each segment consists of one or more alphanumeric characters, dashes, or underscores. Some *Abuild.conf* files exist just to connect parent directories with child directories in the file system. In those cases, the **name** key may be omitted. The **name** key is also optional for root build items that don't build anything themselves.

platform-types
This key is used to specify which platform types a given build item is expected to work on. It includes a whitespace-separated list of platform types. For details about platform types, see Chapter 5, *Target Types, Platform Types, and Platforms*, page 24. If a build item has a build file or an interface file, the **platform-types** key is mandatory. Otherwise, it must not be present. Note that a build item may have multiple platform types, but all platform types for a given build item must belong to the same target type.

plugins
This key is valid only in a root build item. It is used to specify the list of build items that are treated plugins by this tree. For information about plugins, see Chapter 29, *Enhancing Abuild with Plugins*, page 185. A plugin name may be followed by the option **-global** which makes it apply to all build trees in the forest. Use this feature very sparingly. For details, see Section 29.2, “Global Plugins”, page 186.

supported-flags
This key contains a list of whitespace-separated flags that are supported by this build item. When a flag is listed here, it becomes available to this item's *Abuild.interface* file for flag-specific variable assignments. Other items can specify that this flag should be turned on when they depend on this item by using the **-flag=*interface-flag*** option in their **deps** key. For more information, see Chapter 23, *Interface Flags*, page 150.

supported-traits
This key is allowed only in a root build item. It contains a list of whitespace-separated traits that are supported by build items in the build tree. For more information about traits, see Section 9.5, “Traits”, page 42.

traits
This key contains a list of whitespace-separated traits that apply to this build item. A trait may be referent to one or more additional build items. To name a referent build item, follow the trait with the **-item=*build-item*** option. For more information about traits, see Section 9.5, “Traits”, page 42.

tree-deps
This key is valid only in a root build item. It contains a list of the names of trees on which this tree depends. For information about tree dependencies, see Chapter 7, *Multiple Build Trees*, page 33. Tree dependencies may be declared optional by following the name of the dependency with **-optional** (see Chapter 28, *Optional Dependencies*, page 181).

tree-name
The presence of this key establish a build item as a root build item. This key's value is the name of the build tree. Build trees must be named uniquely in a forest. Build tree names may consist of alphanumeric characters, underscore, dash, and period. Unlike with build item names, there is no hierarchical or scoping structure implied by any of the characters in the names of build trees.

visible-to
This key's value is an indicator of the scope at which this build item is visible. If present, it allows build items in the named scope to access this build item directly when they would ordinarily be prevented from doing so by normal scoping rules. For information about build item name scopes and build item visibility, see Section 6.3,

"Build Item Name Scoping", page 28. For a discussion of the **visible-to** key in particular, see Chapter 25, *Build Item Visibility*, page 166

Note that the **child-dirs** keys is the only key that deals with paths rather than names.

Chapter 16. The *Abuild.backing* File

The *Abuild.backing* file may appear at the root of a build forest. It specifies the locations of one or more backing areas and, optionally, provides a list of build items a trees that should not be inherited from the backing areas. For details about backing areas, see Chapter 11, *Backing Areas*, page 59.

The syntax of the *Abuild.backing* file is identical to that of the *Abuild.conf* file: it contains a list of colon-separated key/value pairs. Blank lines and lines beginning with the # character are ignored.

The following keys are defined:

backing-areas
: This key's value is a space-separated list of relative or absolute paths to other build forests that are to be used as a backing area to the current forest. It is the only required key in the *Abuild.backing* file.

deleted-items
: This key's value is a space-separated list of build items that should not be inherited from the backing area. Any build item listed here is treated as an unknown build item in the local forest.

deleted-trees
: This key's value is a space-separated list of build trees that should be inherited from the backing area. Any build item in any build tree listed here will not be made available from the backing area, and the build tree will not be considered a member of the local forest. Note that, unlike with deleted items, it is permissible to create a new build tree locally with the same name as a deleted tree. The new tree is not related to the old tree in any way, and the new tree will not inherit build items from an instance of the deleted tree in the backing areas.

Chapter 17. The Abuild Interface System

The abuild interface system is the mechanism through which abuild provides encapsulation. Its purpose is to allow build items to provide information about the products they provide to other build items. Build items provide their interfaces with the *Abuild.interface* file. This chapter describes the interface system and provides details about the syntax and semantics of *Abuild.interface* and other abuild interface files.

17.1. Abuild Interface Functionality Overview

This section contains a prose description of the interface system's functionality and presents the basic syntax of *Abuild.Interface* without providing all of the details. This material provides the basis for understanding how the interface functionality works. In the next section, we go over the details.

The *Abuild.interface* file has a fairly simple syntax that supports variable declarations, variable assignments, and conditionals. Interface files are rigorously validated. Any errors detected in an interface file are considered build failures which, as such, will prevent abuild from attempting to build the item with the incorrect interface and any items that depend on it. Most *Abuild.interface* files will just set existing variables to provide specific information about that item's include and library information, classpath information, or whatever other standard information may be needed depending upon the type of item it is. For casual users, a full understanding of this material is not essential, but for anyone trying to debug interface issues or create support within abuild for more complex cases, it will be important to understand how abuild reads *Abuild.interface* files.

The basic purpose of *Abuild.interface* is to set variables that are ultimately used by a build item to access its dependencies. The basic model is that an item effectively reads the *Abuild.interface* files of all its dependencies in dependency order. (This is not exactly what happens. For the full story, see Section 33.7, “Implementation of the Abuild Interface System”, page 220.) As each file is read, it adds information to the lists of include paths, libraries, library directories, compiler flags, classpath, etc. All variables referenced by *Abuild.interface* are global variables, even if they are declared inside the body of a conditional, much as is the case with shell scripts or makefiles. Although this is not literally what happens, the best way to think about how abuild reads interface files is to imagine that, for each build item, all of the interface files for its dependencies along with its own interface file are concatenated in dependency order and that the results of that concatenation are processed from top to bottom, skipping over any blocks inside of false conditional statements.

Once abuild parses the *Abuild.interface* files of all of a build item's dependencies and that of the build item itself, the names and values of the resulting variables are passed to the backends by writing them to the abuild *dynamic output file*, which is called *.ab-dynamic.mk* for make-based builds and *.ab-dynamic.groovy* for Groovy/ant-based builds. The dynamic output file is created in the output directory. Although users running abuild don't even have to know this file exists, peeking at it is a useful way to see the results of parsing all the *Abuild.interface* files in a build item's dependency chain.

The *Abuild.interface* file contains the following items:

- Comments
- Variable declarations
- Variable assignments
- After-build file specifications
- Target type restrictions

- Conditionals

Similar to make or shell script syntax, each statement is terminated by the end of the line. Whitespace characters (spaces or tabs) are used to separate words. A backslash (\) as the last character of the line may be used to continue long statements onto the next line of the file, in which case the newline is treated as a word delimiter like any other whitespace. [1] Any line that starts with a # character optionally preceded by whitespace is ignored entirely. Comment lines have no effect on line continuation. In other words, if line one ends with a continuation character and line two is a comment, line one is continued on line three. This makes it possible to embed comments in multiline lists of values. In this example, the value of *ODDS* would be `one three`:

```
ODDS = \
  one \
# odd numbers only, please
  # two \
  three
```

Characters that have special meanings (space, comma, equal, etc.) may be quoted by preceding them by a backslash. For consistency, a backslash followed by any character is treated as that character. This way, the semantics of backslash quoting won't change if additional special characters are added in the future.

All variables must be declared, though most *Abuild.interface* files will be assigning to variables that have already been declared in other interface files. There are no variable scoping rules: all variables are global, even if declared inside a conditional block. Variable names may contain alphanumeric characters, dash, underscore, and period. By convention, make-based rules use all uppercase letters in variable names. This convention also has the advantage of avoiding potential conflict with reserved statements. Java-based rules typically use lower-case period-separated properties. Ultimately abuild interface variables become make variables or ant properties and keys in parameter tables for Groovy, which is the basis for these conventions. Note, however, that variables of both naming styles may be used by either backend, and some of abuild's predefined interface variables that are available to both make and Groovy/ant are of the all upper-case variety.

Once declared, a variable may be assigned to or referenced. A variable is referenced by enclosing its name with parentheses and preceding it by a dollar sign (as in *$(VARIABLE)*), much like with standard make syntax, except that there is no special case for single-character variable names. Other than using the backslash character to quote single characters, there is no quoting syntax: the single and double quote characters are treated as ordinary characters with no special meanings.

Environment variables may be referenced using the syntax *$(ENV:VARIABLE)*. Unlike many other systems which treat undefined environment variables as the empty string, abuild will trigger an error condition if the environment variable does not exist unless a default value is provided. A default value can be provided using the syntax *$(ENV:VARIABLE:default-value)*. The *`default-value`* portion of the string may not contain spaces, tabs, or parentheses. [2] Although it can sometimes be useful to have abuild interface files initialize interface variables from the environment, this feature should be used sparingly as it is possible to make a build become overly dependent on the environment in this way. (Even without this feature, there are other ways to fall into this trap that are even worse.) Note that environment variables are not abuild variables. They are expanded as strings and can be used in the interface file wherever ordinary strings can be used.

In addition, starting in version 1.1.1, abuild can access command-line parameters of the form *`VAR=val`* from interface files. This works identically to environment variables. Parameter references are of the form *$(PARAM:PARAMETER)*

[1] In this way, abuild's handles line continuation like GNU Make and the C shell. This is different from how the Bourne shell and the C programming language treat line continuation characters: in those environments, a quoted newline disappears entirely. The only time this matters is if there are no spaces at the beginning of a line following a line continuation character. For abuild, make, and the C shell it doesn't matter whether or not space is present at the beginning of a line following a line continuation character, but for C and the Bourne shell, it does.

[2] This syntax restriction is somewhat arbitrary, but it makes it less likely that syntax errors in specifying environment variable references will create hard-to-solve parsing errors in interface files. If this restriction is in your way, you're probably abusing this feature and may need to rethink why you're accessing environment variables to begin with.

or *$(PARAM:PARMETER:default-value)*. As with environment variable references, accessing an unspecified parameter without a default is an error, and parameter expansions are treated as strings by the interface parser. This feature should also be used sparingly as it can create plenty of opportunity for unpredictable builds. The main valid use case for accessing parameters from an interface file would be to allow special debugging changes that allow modifying build behavior from the command-line for particular circumstances. Keep in mind that changing parameters on the command line has no impact on dependencies, so gratuitous and careless use of this feature can lead to unreproducible builds. That said, this feature does not make abuild inherently less safe since it has always been possible to access parameters and the environment directly from make code.

Variables may contain single scalar values or they may contain lists of values of one of the three supported types: *boolean*, *string*, or *filename*.

Boolean variables are simple true/false values. The values `1` and `true` are interpreted interchangeably as true, and the values `0` and `false` are interpreted interchangeably as false. Regardless of whether the word or numeric value is used to assign to boolean variables, the normalized values of `0` and `1` are passed to the backend build system. (For simplicity and consistency, this is true even for the Groovy backend, which could handle actual boolean values instead.) String variables just contain arbitrary text. It is possible to embed spaces in string variables by quoting them with a backslash, but keep in mind that not all backends handle spaces in single-word variable values cleanly. For example, dealing with embedded spaces in variable names in GNU Make is impractical since it uses space as a word delimiter and offers no specific quoting mechanisms. The values of filename variables are interpreted to be path names. Path names may be specified with either forward slashes or backslashes on any platform. Relative paths (those that do not start with a path separator character or, on Windows, also a drive letter) are interpreted as *relative to the file in which they are assigned*, not the file in which they are referenced as is the case with make. This means that build items can export information about their local files using relative paths without having to use any special variables that point to their own local directories. Although this is different from how make works, it is the only sensible semantic for files that are referenced from multiple locations, and it is one of the most important and useful features of the abuild interface system.

List variables may contain multiple space-separated words. Assignments to list variables may span multiple lines by using a trailing backslash to indicate continuation to the next line. Each element of a list must be the same type. Lists can be made of any of the supported scalar types. (Lists of boolean values are supported, though they are essentially useless.) List variables must be declared as either *append* or *prepend*, depending upon whether successive assignments are appended or prepended to the value of the list. This is described in more depth when we discuss variable assignment below.

Scalar variables may be assigned to in one of three ways: *normal*, *override*, and *fallback*. A normal assignment to a scalar variable fails if the variable already has a value. An override assignment initializes a previously uninitialized variable and replaces any previously assigned value. A fallback assignment sets the value of the variable only if it has not previously been initialized. Uninitialized variables are passed to the backend as empty strings. It is legal to initialize a string variable to the empty string, and doing this is distinct from not initializing it.

List variables work differently from anything you're likely to have encountered in other environments, but they offer functionality that is particularly useful when building software. List variables may be assigned to multiple times. The value in each individual assignment may contain zero or more words. Depending on whether the variable was declared as *append* or *prepend*, the values are appended to or prepended to the list in the order in which they appear in the specific assignment. An example is provided below.

Scalar and list variables can both be reset using the *reset* statement. This resets the variable back to its initial state, which is uninitialized for scalars and empty for lists.

Any variable assignment statement can be made conditional upon the presence of a given interface flag. Interface flags are introduced in Chapter 23, *Interface Flags*, page 150, and the details of how to use them in interface files are discussed later in this chapter.

Abuild supports nested conditionals, each of which may contain an *if* clause, zero or more *elseif* clauses, and an optional *else* clause. The abuild interface syntax supports no relational operators: all conditionals are expressed in terms of function calls, the details of which are provided below.

In addition to supporting variables and conditionals, it is possible to specify that certain variables are relevant only to build items of a specific target type. A target type restriction applies until the next *target-type* directive or until the end of the current file and all the files it loads as *after-build* files. By default, declarations in an *Abuild.interface* file apply to all target types. The vast majority of interface files will not have to include any target type restrictions.

It is possible for a build item to contain interface information that is intended for items that depend on it but not intended for the item itself. Typical uses cases would include when some of this information is a product of the build or when a build item needs to modify interface information provided by a dependency after it has finished using the information itself. To support this, an *Abuild.interface* file may specify additional interface files that are not to be read until after the item is built. The values in any such files are not available to the build item itself, but they are available to any items that depend on the build item that exports this interface. Such files may be dynamically generated (such as with autoconf; see Section 18.3, “Autoconf Example”, page 99), or they may be hand-generated files that are just intended not to apply to the build of the current build item (see Section 27.1, “Opaque Wrapper Example”, page 179).

By default, once a variable is declared and assigned to in a build item's *Abuild.interface*, the declaration and assignments are automatically visible to all build items that depend on the item that made the declaration or assignment. In this sense, abuild variables are said to be *recursive*. It is also possible to declare a variable as *non-recursive*, in which case assignments to the variable are only visible in the item itself and in items that depend *directly* on the item that makes the assignment. Declarations inherit normally. [3]

It is also possible to declare an interface variable as *local*. When a variable is declared as local, the declaration and assignment are not visible to any other build items. This can be useful for providing values only to the current build item or for using variables to hold temporary values within the *Abuild.interface* file and any after-build files that it may explicitly reference.

17.2. Abuild.interface Syntactic Details

In this section, we provide the syntactic details for each of the capabilities described in the previous section. There are some aspects of how *Abuild.interface* files are interpreted that are different from other systems you have likely encountered. If you are already familiar with the basics of how these files work, this section can serve as a quick reference.

Note

If you only read one thing, read about list assignment. Assignment to list variables is probably different for *Abuild.interface* files than for any other variable assignment system you're likely to have encountered. It is specifically designed to support building up lists gradually by interpreting multiple files in a specific order.

comment
: Any line beginning with a # optionally preceded by whitespace is treated as a comment. Comments are completely ignored and, as such, have no effect on line continuation. Note that the # does not have any special meaning when it appears in another context. There is no syntax for including comments within a line that contains other content.

variable declaration
: A scalar variable declaration takes the form

```
declare variable [ scope ] type [ = value ]
```

[3] The rationale behind using the terms *recursive* and *non-recursive* have to do with how these variables are used. Conceptually, when you reference an interface variable, you see all assignments made to it by any of your *recursively expanded* list of dependencies, *i.e.*, your direct and indirect dependencies. When a variable is declared to be non-recursive, you only assignments made by your direct dependencies. Other terms, such as *indirect* or *non-inheriting* would be technically incorrect or slightly misleading. Although there's nothing specifically recursive or non-recursive about how interface variables are used, we feel that this choice of terminology is a reasonable reflection of the semantics achieved.

where *variable* is the name of the variable and *type* is one of boolean, string, or filename. If specified, *scope* may be one of non-recursive or local. The declaration may also be followed optionally be an initialization, which takes the same form as assignment, described below. Example scalar variable declarations:

```
declare CODEGEN filename
declare HAS_CLASS boolean
declare _dist local filename = $(ABUILD_OUTPUT_DIR)/dist
```

A list variable declaration takes the form

```
declare variable [ scope ] list type append-type [ = value ]
```

where *variable* is the name of the variable, *type* is one of boolean, string, or filename, and *append-type* is one of append or prepend. The optional *scope* specification is the same as for scalar variables (non-recursive or local), and as with scalar variables, an optional initialization may be provided. Example list variable declarations:

```
declare QFLAGS list string append
declare QPATHS list filename prepend = qfiles private-qfiles
declare DEPWORDS non-recursive list string append
```

Scalar variables start off uninitialized. List variables start off containing zero items.

scalar variable assignment

Scalar variables may be assigned in one of three ways: normal, override, or default. A normal assignment looks like this:

```
variable = value
```

where *variable* is the variable name and *value* is a single word (leading and trailing space ignored). Extra whitespace is permitted around the = sign.

Override assignments look like this:

```
override variable = value
```

Fallback assignments look like this:

```
fallback variable = value
```

Example scalar variable assignments:

```
fallback CODEGEN = gen_code.pl
HAS_CLASS = 0
override HAS_CLASS = 1
```

list variable assignment

List variables are assigned using a simple = operator:

```
list-variable = value
```

where *value* consists of zero or more words, and the semantics of the assignment depend on how the list was declared. For append lists, the assignment operator appends the words to the existing list in the order in which