The `build2` Toolchain Introduction

Revision `0.7`, May 2018
This revision of the document describes the `build2` toolchain `0.7.x` series.

Table of Contents

Preface

This document is an overall introduction to the build2 toolchain that shows how the main components, namely the build system, the package dependency manager, and the project dependency manager are used together to handle the entire C++ project development lifecycle: creation, development, testing, and delivery. For additional information, including documentation for individual toolchain components, man pages, etc., refer to the build2 project Documentation page.

1 TL;DR

```
$ git clone ssh://example.org/hello.git
$ tree hello
hello/
âââ hello/
âÂ Â  âââ hello.cxx
âÂ Â  âââ buildfile
âââ manifest
âââ repositories.manifest

$ cd hello
$ bdep init --config-create ../hello-gcc cc config.cxx=g++
initializing in project /tmp/hello/
created configuration /tmp/hello-gcc/ default,auto-synchronized
synchronizing:
  new hello/0.1.0

$ b
c++ hello/cxx{hello}@../hello-gcc/hello/hello/
ld ../hello-gcc/hello/hello/exe{hello}
ln ../hello-gcc/hello/hello/exe{hello} -> hello/

$ hello/hello World
Hello, World!

$ edit repositories.manifest   # add https://example.org/libhello.git
$ edit manifest                # add 'depends: libhello ^1.0.0'
$ edit hello/buildfile         # import libhello
$ edit hello/hello.cxx         # use libhello

$ b
fetching from https://example.org/libhello.git
synchronizing /tmp/hello-gcc/:
  new libhello/1.0.0 (required by hello)
  reconfigure hello/0.1.0
c++ ../hello-gcc/libhello-1.0.0/libhello/cxx{hello}
ld ../hello-gcc/libhello-1.0.0/libhello/libs{hello}
c++ hello/cxx{hello}@../hello-gcc/hello/hello/
ld ../hello-gcc/hello/hello/exe{hello}
ln ../hello-gcc/hello/hello/exe{hello} -> hello/

$ bdep fetch                   # refresh available versions
```

```
$ bdep status -i               # review available versions
hello configured 0.1.0
  libhello ^1.0.0 configured 1.0.0 available [1.1.0]

$ bdep sync libhello           # upgrade to latest
synchronizing:
  new libformat/1.0.0 (required by libhello)
  new libprint/1.0.0 (required by libhello)
  upgrade libhello/1.1.0
  reconfigure hello/0.1.0

$ bdep sync libhello/1.0.0     # downgrade
synchronizing:
  drop libprint/1.0.0 (unused)
  drop libformat/1.0.0 (unused)
  downgrade libhello/1.0.0
  reconfigure hello/0.1.0
```

2 Getting Started Guide

The aim of this guide is to get you started developing C/C++ projects with the `build2` toolchain. All the examples in this section include the relevant command output so if you just want to get a sense of what `build2` is about, then you don't have to install the toolchain and run the commands in order to follow along. Or, alternatively, you can take a short detour to the Installation Instructions and then try the examples for yourself.

One of the primary goals of the `build2` toolchain is to provide a uniform interface across all the platforms and compilers. While most of the examples in this document assume a UNIX-like operation system, they will look pretty similar if you are on Windows. You just have to use appropriate paths, compilers, and options.

The question we will try to answer in this section can be summarized as:

```
$ git clone .../hello.git && now-what?
```

That is, we clone an existing C/C++ project or would like to create a new one and then start hacking on it. We want to spend as little time and energy as possible on the initial and ongoing infrastructure maintenance: setting up build configurations, managing dependencies, continuous integration and testing, release management, etc. Or, as one C++ user aptly put it, "*All I want to do is program.*"

2.1 Hello, World

Let's see what programming with `build2` feels like by starting with a customary *"Hello, World!"* program (here we assume our current working directory is `/tmp`):

```
$ bdep new -t exe -l c++ hello
created new executable project hello in /tmp/hello/
```

The **bdep-new(1)** command creates a *canonical* `build2` project. In our case it is an executable implemented in C++.

To create a library, pass `-t lib`. By default `new` also initializes a `git` repository and generates suitable `.gitignore` files (pass `-s none` if you don't want that). And for details on naming your projects, see Package Name.

Note to Windows users: the `build2-baseutils` package includes core `git` utilities that are sufficient for the `bdep` functionality.

Let's take a look inside our new project:

```
$ tree hello
hello/
âââ .git/
âââ .bdep/
âââ build/
âââ hello/
â   âââ hello.cxx
â   âââ buildfile
â   âââ testscript
âââ buildfile
âââ manifest
âââ repositories.manifest
```

While the canonical project structure is strongly recommended, especially for new projects, `build2` is flexible enough to allow most commonly used arrangements.

Similar to version control tools, we normally run all `build2` tools from the project's source directory or one of its subdirectories, so:

```
$ cd hello
```

While the project layout is discussed in more detail in later sections, let's examine a couple of interesting files to get a sense of what's going on. We start with the source file which should look familiar:

```
$ cat hello/hello.cxx

#include <iostream>

int main (int argc, char* argv[])
{
  using namespace std;

  if (argc < 2)
  {
    cerr << "error: missing name" << endl;
```

```
    return 1;
  }

  cout << "Hello, " << argv[1] << '!' << endl;
}
```

If you prefer the `.?pp` extensions over `.?xx` for your C++ source files, pass `-l c++,cpp` to the `new` command. See **`bdep-new(1)`** for details on this and other customization options.

Let's take a look at the accompanying `buildfile`:

```
$ cat hello/buildfile

libs =
#import libs += libhello%lib{hello}

exe{hello}: {hxx ixx txx cxx}{*} $libs testscript
```

As the name suggests, this file describes how to build things. While its content might look a bit cryptic, let's try to infer a couple of points without going into too much detail (the details are discussed in the following sections). That `exe{hello}` on the left of `:` is a *target* (executable named `hello`) and what we have on the right are *prerequisites* (C++ source files, libraries, etc). This `buildfile` uses wildcard patterns (that `*`) to automatically locate all the C++ source files. This means we don't have to edit our `buildfile` every time we add a source file to our project. There also appears to be some (commented out) infrastructure for importing and linking libraries (that `libs` variable). We will see how to use it in a moment. Finally, the `buildfile` also lists `testscript` as a prerequisite of `hello`. This file tests our target. Let's take a look inside:

```
$ cat hello/testscript

: basics
:
$* 'World' >'Hello, World!'

: missing-name
:
$* 2>>EOE != 0
error: missing name
EOE
```

Again, we are not going into detail here (see Testscript Introduction for a proper introduction), but to give you an idea, here we have two tests: the first (with id `basics`) verifies that our program prints the expected greeting while the second makes sure it handles the missing name error condition. Tests written in Testscript are concise, portable, and executed in parallel.

Next up is `manifest`:

```
$ cat manifest
: 1
name: hello
version: 0.1.0-a.0.z
summary: hello executable
license: TODO
url: https://example.org/hello
email: you@example.org
#depends: libhello ^1.0.0
```

The `manifest` file is what makes a build system project a *package*. It contains all the metadata that a user of a package might need to know: its name, version, license, dependencies, etc., all in one place.

Refer to Manifest Format for the general format of `build2` manifest files and to Package Manifest for details on the package manifest values.

As you can see, `manifest` created by **`bdep-new(1)`** contains some dummy values which you would want to adjust before publishing your package. But let's resist the urge to adjust that strange looking `0.1.0-a.0.z` until we discuss package versioning.

Next to `manifest` you might have noticed the `repositories.manifest` file – we will discuss its function later, when we talk about dependencies and where they come from.

Project in hand, let's build it. Unlike other programming languages, C++ development usually involves juggling a handful of build configurations: several compilers and/or targets (`build2` is big on cross-compiling), debug/release, different sanitizers and/or static analysis tools, and so on. As a result, `build2` is optimized for multi-configuration usage. However, as we will see shortly, one build configuration can be designated as the default with additional conveniences.

The **`bdep-init(1)`** command is used to initialize a project in a build configuration. As a shortcut, it can also create a new build configuration in the process, which is just what we need here. Let's start with GCC (remember we are in the project's root directory):

```
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++
initializing in project /tmp/hello/
created configuration @gcc /tmp/hello-gcc/ default,auto-synchronized
synchronizing:
  new hello/0.1.0-a.0.19700101000000
```

The **`--create|-C`** option instructs `init` to create a new configuration in the specified directory (`../hello-gcc` in our case). To make referring to configurations easier, we can give it a name, which is what we do with `@gcc`. The next argument (`cc`, stands for *C-common*) is the build system module we would like to configure. It implements compilation and linking rules for the C and C++ languages. Finally, `config.cxx=g++` is (one of) this module's configuration variables that specifies the C++ compiler we would like to use (the corresponding C compiler will be determined automatically). Let's for now also ignore that `synchronizing:...` bit along with strange-looking `19700101000000` in the version – it will become clear what's going on here in a moment.

Note to Windows users: a command line argument with leading `@` has a special meaning in PowerShell. To work around this, you can use the alternative `-@gcc` syntax or the `-n gcc` option.

Now the same for Clang:

```
$ bdep init -C ../hello-clang @clang cc config.cxx=clang++
initializing in project /tmp/hello/
created configuration @clang /tmp/hello-clang/ auto-synchronized
synchronizing:
  new hello/0.1.0-a.0.19700101000000
```

If we check the parent directory, we should now see two build configurations next to our project:

```
$ ls ..
hello/
hello-gcc/
hello-clang/
```

If, as in the above examples, our configuration directories are next to the project and their names are in the *`prj-name`*`-`*`cfg-name`* form, then we can use the shortcut version of the `init` command:

```
$ bdep init -C @clang cc config.cxx=clang++
```

Things will also look pretty similar if you are on Windows instead of a UNIX-like operating system. For example, to initialize our project on Windows with Visual Studio, start the Visual Studio development command prompt and then run:

Currently we have to run `build2` tools from a suitable Visual Studio development command prompt. This requirement will likely be removed in the future.

```
> bdep init -C ..\hello-debug @debug cc     ^
  config.cxx=cl                              ^
  "config.cc.coptions=/MDd /Z7"              ^
  config.cc.loptions=/DEBUG

> bdep init -C ..\hello-release @release cc ^
  config.cxx=cl                              ^
  config.cc.coptions=/O2
```

Besides the `coptions` (compile options) and `loptions` (link options), other commonly used `cc` module configuration variables are `poptions` (preprocess options) and `libs` (extra libraries to link). We can also use their `config.c.*` (C compilation) and `config.cxx.*` (C++ compilation) variants if we only want them applied during the respective language compilation. For example:

```
$ bdep init ... cc       \
  config.cxx=clang++     \
  config.cc.coptions=-g  \
  config.cxx.coptions=-stdlib=libc++
```

One difference you might have noticed when creating the `gcc` and `clang` configurations above is that the first one was designated as the default. The default configuration is used by `bdep` commands if no configuration is specified explicitly (see **`bdep-projects-configs(1)`** for details). It is also the configuration that is used if we run the build system in the project's source directory. So, normally, you would make your every day development configuration the default. Let's try that:

```
$ bdep status
hello configured 0.1.0-a.0.19700101000000

$ b
c++ hello/cxx{hello}@../hello-gcc/hello/hello/
ld ../hello-gcc/hello/hello/exe{hello}
ln ../hello-gcc/hello/hello/exe{hello} -> hello/

$ b test
test hello/test{testscript} ../hello-gcc/hello/hello/exe{hello}

$ hello/hello World
Hello, World!
```

To see the actual compilation command lines, run `b -v` and for even more details, run `b -V`. See **`b(1)`** for more information on these and other build system options.

In contrast, the Clang configuration has to be requested explicitly:

```
$ bdep status @clang
hello configured 0.1.0-a.0.19700101000000

$ b ../hello-clang/hello/
c++ hello/cxx{hello}@../hello-clang/hello/hello/
ld ../hello-clang/hello/hello/exe{hello}

$ b test: ../hello-clang/hello/
test hello/test{testscript} ../hello-clang/hello/hello/exe{hello}

$ ../hello-clang/hello/hello/hello World
Hello, World!
```

As you can see, using the build system directly on configurations other than the default requires explicitly specifying their paths. It would have been more convenient if we could refer to them by names. The **`bdep-update(1)`** and **`bdep-test(1)`** commands allow us to do exactly that:

```
$ bdep test @clang
c++ hello/cxx{hello}@../hello-clang/hello/hello/
ld ../hello-clang/hello/hello/exe{hello}
test hello/test{testscript} ../hello-clang/hello/hello/exe{hello}
```

And we can also perform the desired build system operation on several (or `--all|-a`) configurations at once:

```
$ bdep test @gcc @clang
in configuration @gcc:
test hello/test{testscript} ../hello-gcc/hello/hello/exe{hello}

in configuration @clang:
test hello/test{testscript} ../hello-clang/hello/hello/exe{hello}
```

As we will see later, the **`bdep-test(1)`** command also allows us to test immediate (`--immediate|-i`) or all (`--recursive|-r`) dependencies of our project.

While we are here, let's also check how hard it would be to cross-compile:

```
$ bdep init -C ../hello-mingw @mingw cc config.cxx=x86_64-w64-mingw32-g++
initializing in project /tmp/hello/
created configuration @mingw /tmp/hello-mingw/ auto-synchronized
synchronizing:
  new hello/0.1.0-a.0.19700101000000

$ bdep update @mingw
c++ hello/cxx{hello}@../hello-mingw/hello/hello/
ld ../hello-mingw/hello/hello/exe{hello}
```

As you can see, cross-compiling in `build2` is nothing special. In our case, on a properly setup GNU/Linux machine (that automatically uses `wine` as an `.exe` interpreter) we can even run tests (in `build2` this is called *cross-testing*):

```
$ bdep test @mingw
test hello/test{testscript} ../hello-mingw/hello/hello/exe{hello}

$ ../hello-mingw/hello/hello/hello.exe Windows
Hello, Windows!
```

Let's review what it takes to initialize a project's infrastructure and perform the first build. For an existing project:

```
$ git clone .../hello.git
$ cd hello
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++
$ b
```

For a new project:

```
$ bdep new -t exe -l c++ hello
$ cd hello
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++
$ b
```

If you prefer, the `new` and `init` steps can be combined into a single command:

```
$ bdep new -t exe -l c++ hello -C hello-gcc @gcc cc config.cxx=g++
```

And if you need to deinitialize a project in one or more build configurations, there is the **`bdep-deinit(1)`** command for that:

```
$ bdep deinit @gcc @clang
deinitializing in project /tmp/hello/
in configuration @gcc:
synchronizing:
  drop hello

in configuration @clang:
synchronizing:
  drop hello
```

Now is also a good time to get an overview of the `build2` toolchain. After all, we have already used two of its tools (`bdep` and `b`) without a clear understanding of what they actually are.

Unlike most other programming languages that encapsulate the build system, package dependency manager, and project dependency manager into a single tool (such as Rust's `cargo` or Go's `go`), `build2` is a hierarchy of several tools that you will be using directly and which together with your version control system (VCS) will constitute the core of your project management toolset.

While `build2` can work without a VCS, this will result in reduced functionality.

At the bottom of the hierarchy is the build system, **`b(1)`**. Next comes the package dependency manager, **`bpkg(1)`**. It is primarily used for *package consumption* and depends on the build system. The top of the hierarchy is the project dependency manager, **`bdep(1)`**. It is used for *project development* and relies on `bpkg` for building project packages and their dependencies.

The main reason for this separation is modularity and the resulting flexibility: there are situations where we only need the build system (for example, when building a package for a system package manager where all the dependencies should be satisfied from the system repository), or only the build system and package manager (for example, when a build bot is building a package for testing).

Note also that strictly speaking `build2` is not C/C++-specific; its build model is general enough to handle any DAG-based operations and its package/project dependency management can be used for any compiled language.

As we will see in a moment, `build2` also integrates with your VCS in order to automate project versioning. Note that currently only `git(1)` is supported.

Now that we understand the tooling, let's also revisit the notion of *build configuration* (those `hello-gcc` and `hello-clang` directories). A `bdep` build configuration is actually a `bpkg` build configuration which, in the build system terms, is an *amalgamation* – a project that contains *subprojects*. In our case, the subprojects in these amalgamations will be the projects we have initialized with `init` and, as we will see later, packages that they depend on. For example, here is what our `hello-gcc` contains:

```
$ tree hello-gcc
hello-gcc/
âââ .bpkg/
âââ build/
âÂ Â  âââ config.build
âââ hello/
    âââ build/
    âÂ Â  âââ config.build
    âââ hello/
        âââ hello
        âââ hello.o
```

Underneath **`bdep-init(1)`** with the `--config-create|-C` option calls **`bpkg-cfg-create(1)`** which, in turn, performs the build system `create` meta-operation (see **`b(1)`** for details).

The important point here is that the `bdep` build configuration is not a black box that you should never look inside of. On the contrary, it is a normal and predictable concept of the package manager and the build system and as long as you understand what you are doing, you should feel free to interact with it directly.

Let's now move on to the reason why there is *dep* in the `bdep` name: dependency management.

2.2 Package Repositories

Say we have realized that writing *"Hello, World!"* programs is a fairly common task and that someone must have written a library to help with that. So let's see if we can find something suitable to use in our project.

Where should we look? That's a good question. But before we can try to answer it, we need to understand where `build2` can source dependencies. In `build2` packages come from *package repositories*. Two commonly used repository types are *version control* and *archive*-based (see **`bpkg-repository-types(1)`** for details).

As the name suggests, a version control-based repository uses a VCS as its distribution mechanism. Currently, only `git` is supported. Such a repository normally contains multiple versions of a single package or, perhaps, of a few related packages.

An archive-based repository contains multiple, potentially unrelated packages/versions as archives along with some meta information (package list, prerequisite/complement repositories, signatures, etc) that are all accessible via HTTP(S).

Version control and archive-based repositories have different trade-offs. Version control-based repositories are great for package developers: With services like GitHub they are trivial to setup. In fact, your project's (already existing) VCS repository will normally be the `build2` package repository – you might need to add a few files, but that's about it.

However, version control-based repositories are not without drawbacks: It will be hard for your users to discover your packages (try searching for "hello library" on GitHub – most of the results are not even in C++ let alone packaged for `build2`). There is also the issue of continuous availability: users can delete their repositories, services may change their policies or go out of business, and so on. Version control-based repositories also lack repository authentication and package signing. Finally, obtaining the available package list for such repositories can be slow.

A central, archive-based repository would address all these drawbacks: It would be a single place to search for packages. Published packages will never disappear and can be easily mirrored. Packages are signed and the repository is authenticated (see **`bpkg-repository-signing(1)`** for details). And, last, but not least, archive-based repositories are fast.

cppget.org is the `build2` community's central package repository. While centralized, it is also easy to mirror since its contents are accessible via plain HTTPS (you can browse pkg.cppget.org to get an idea). As an added benefit, packages on cppget.org are continuously built and tested on all the major platform/compiler combinations with the results available as part of the package description.

The main drawback of archive-based repositories is the setup cost. Getting a basic repository going is relatively easy – all you need is an HTTP(S) server. Adding a repository web interface like that on cppget.org will require running `brep`. And adding CI will require running a bunch of build bots (`bbot`).

CI support for version control-based repositories is a work in progress.

To summarize, version control-based repositories are great for package developers while a central, archive-based repository is convenient for package consumers. A reasonable strategy is then for package developers to publish their releases to a central repository. Package consumers can then decide which repository to use based on their needs. For example, one could use cppget.org as a (fast, reliable, and secure) source of stable versions but also add, say, `git` repositories for select packages (perhaps with the `#HEAD` fragment filter to improve download speed) for testing development snapshots. In this model the two repository types complement each other.

Support for automated publishing of tagged releases to an archive-based repository is a work in progress.

Let's see how all this works in practice. Go over to cppget.org and type "hello library" in the search box. At the top of the search result you should see the `libhello` package and if you follow the link you will see the package description page along with a list of available versions. Pick a version that you like and you will see the package version description page with quite a bit of information, including the list of platform/compiler combinations that this version has been successfully (or unsuccessfully) tested with. If you like what you see, copy the `location` value – this is the repository location where this package version can be sourced from.

The cppget.org repository is split into several sections: `stable`, `testing`, `beta`, `alpha` and `legacy`, with each section having its own repository location (see the repository's about page for details on each section's policies). Note also that `testing` is complemented by `stable`, `beta` by `testing`, and so on, so you only need to choose the lowest stability level and you will automatically "see" packages from the more stable sections.

The cppget.org `stable` sections will always contain the `libhello` library version `1.0.X` that was generated using the following **`bdep-new(1)`** command line:

```
$ bdep new -t lib -l c++ libhello
```

It can be used as a predictable test dependency when setting up new projects.

Let's say we've visited the `libhello` project's home page (for example by following a link from the package details page) and noticed that it is being developed in a `git` repository. How can we see what's available there? If the releases are tagged, then we can infer the available released versions from the tags. But that doesn't tell us anything about what's happening on the `HEAD` or in the branches. For that we can use the package manager's **`bpkg-rep-info(1)`** command:

```
$ bpkg rep-info https://git.build2.org/hello/libhello.git
libhello/1.0.0
libhello/1.1.0
```

As you can see, besides `1.0.0` that we have seen in `cppget.org/stable`, there is also `1.1.0` (which is perhaps being tested in `cppget.org/testing`). We can also check what might be available on the `HEAD` (see **`bpkg-repository-types(1)`** for details on the `git` repository URL format):

```
$ bpkg rep-info https://git.build2.org/hello/libhello.git#HEAD
libhello/1.1.1-a.0.20180504111511.2e82f7378519
```

We can also use the `rep-info` command on archive-based repositories, however, if available, the web interface is usually more convenient and provides more information.

To summarize, we found two repositories for the `libhello` package: the archive-based cppget.org that contains the released versions as well as its development `git` repository where we can get the bleeding edge stuff. Let's now see how we can add `libhello` to our project.

2.3 Adding and Removing Dependencies

So we found `libhello` that we would like to use in our `hello` project. First, we edit the `repositories.manifest` file found in the root directory of our project and add one of the `libhello` repositories as a prerequisite. Let's start with cppget.org:

```
role: prerequisite
location: https://pkg.cppget.org/1/stable
```

Refer to Repository Manifest for details on the repository manifest values.

Next, we edit the `manifest` file (again, found in the root of our project) and specify the dependency on `libhello` with optional version constraint. For example:

```
depends: libhello ^1.0.0
```

Let's briefly discuss version constraints (for details see the `depends` value documentation). A version constraint can be expressed with a comparison operator (==, >, <, >=, <=), a range shortcut operator (~ and ^), or a range. Here are a few examples:

```
depends: libhello == 1.2.3
depends: libhello >= 1.2.3

depends: libhello ~1.2.3
depends: libhello ^1.2.3

depends: libhello [1.2.3 1.2.9)
```

You may already be familiar with the tilde (~) and caret (^) constraints from dependency managers for other languages. To recap, tilde allows upgrades to any further patch versions while caret also allows upgrades to further minor versions. They are equivalent to the following ranges:

```
~X.Y.Z  [X.Y.Z  X.Y+1.0)

^X.Y.Z  [X.Y.Z  X+1.0.0)  if X >  0
^0.Y.Z  [0.Y.Z  0.Y+1.0)  if X == 0
```

Zero major version component is customarily used during early development where the minor version effectively becomes major. As a result, the tilde constraint has a special treatment of this case.

Unless you have good reasons not to (for example, a dependency does not use semantic versioning), we suggest that you use the ^ constraint which provides a good balance between compatibility and upgradability with ~ being a more conservative option.

Ok, we've specified where our package comes from (`repositories.manifest`) and which versions we find acceptable (`manifest`). The next step is to edit `hello/buildfile` and import the `libhello` library into our build:

```
import libs += libhello%lib{hello}
```

Finally, we modify our source code to use the library:

```
#include <libhello/hello.hxx>
...

int main (int argc, char* argv[])
{
  ...
  hello::say_hello (cout, argv[1]);
}
```

You are probably wondering why we have to specify this repeating information in so many places. Let's start with the source code: we can't specify the version constraint or location there because it will have to be repeated in every source file that uses the dependency.

Moving up, `buildfile` is also not a good place to specify this information for the same reason (a library can be imported in multiple buildfiles) plus the build system doesn't really know anything about version constraints or repositories which is the purview of the dependency management tools.

Finally, we have to separate the version constraint and the location because the same package can be present in multiple repositories with different policies. For example, when a package from a version control-based repository is published in an archive-based repository, its `repositories.manifest` file is ignored and all its dependencies should be available from the archive-based repository itself (or its fixed set of prerequisite repositories). In other words, `manifest` belongs to a package while `repositories.manifest` – to a repository.

Also note that this is unlikely to become burdensome since adding new dependencies is not something that happens often. There are also plans to automate this with a `bdep-add(1)` command in the future.

To summarize, these are the files we had to modify to add a dependency to our project:

```
repositories.manifest   # add https://pkg.cppget.org/1/stable
manifest                # add 'depends: libhello ^1.0.0'
buildfile               # import libhello
hello.cxx               # use libhello
```

With a new dependency added, let's check the status of our project:

```
$ bdep status
fetching pkg:cppget.org/stable (prerequisite of dir:/tmp/hello)
warning: authenticity of the certificate for pkg:cppget.org/stable
         cannot be established
certificate is for cppget.org, "Code Synthesis" <admin@cppget.org>
certificate SHA256 fingerprint:
86:BA:D4:DE:2C:87:1A:EE:38:<...>:5A:EA:F4:F7:8C:1D:63:30:C6
trust this certificate? [y/n] y

hello configured 0.1.0-a.0.19700101000000
      available  0.1.0-a.0.19700101000000#1
```

The **bdep-status(1)** command has detected that the dependency information has changed and tells us that a new *iteration* of our project (that `#1`) is now available for *synchronization* with the build configuration.

We've also been prompted to authenticate the prerequisite repository. This will have to happen once for every build configuration we initialize our project in and can quickly become tedious. To overcome this, we can mention the certificate fingerprint that we wish to automatically trust in the `repositories.manifest` file (replace it with the actual fingerprint from the repository's about page):

```
role: prerequisite
location: https://pkg.cppget.org/1/stable
trust: 86:BA:D4:DE:2C:87:1A:EE:38:<...>:5A:EA:F4:F7:8C:1D:63:30:C6
```

To synchronize a project with one or more build configurations we use the **bdep-sync(1)** command:

```
$ bdep sync
synchronizing:
  new libhello/1.0.0 (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#1
```

Or we could just build the project without an explicit `sync` – if necessary, it will be automatically synchronized:

```
$ b
synchronizing:
  new libhello/1.0.0 (required by hello)
  upgrade hello/0.1.0-a.0.19700101000000#1
c++ ../hello-gcc/libhello-1.0.0/libhello/cxx{hello}
ld ../hello-gcc/libhello-1.0.0/libhello/libs{hello}
c++ hello/cxx{hello}@../hello-gcc/hello/hello/
ld ../hello-gcc/hello/hello/exe{hello}
ln ../hello-gcc/hello/hello/exe{hello} -> hello/
```

The synchronization as performed by the `sync` command is two-way: dependency packages are first added, removed, upgraded, or downgraded in build configurations according to the project's version constraints and user input. Then the actual versions of the dependencies present in the build configurations are recorded in the project's `lockfile` so that if desired, the build can be reproduced exactly. The `lockfile` functionality is not yet implemented. For a new dependency the latest available version that satisfies the version constraint is used.

Synchronization is also the last step in the **`bdep-init(1)`** command's logic.

Let's now examine the status in all (`--all|-a`) the build configurations and include the immediate dependencies (`--immediate|-i`):

```
$ bdep status -ai
in configuration @gcc:
hello configured 0.1.0-a.0.19700101000000#1
  libhello ^1.0.0 configured 1.0.0

in configuration @clang:
hello configured 0.1.0-a.0.19700101000000
      available 0.1.0-a.0.19700101000000#1
```

Since we didn't specify a configuration explicitly, only the default (`gcc`) was synchronized. Normally, you would try a new dependency in one configuration, make sure everything looks good, then synchronize the rest with `--all|-a` (or, again, just build what you need directly). Here are a few examples (see **`bdep-projects-configs(1)`** for details):

```
$ bdep sync -a
$ bdep sync @gcc @clang
$ bdep sync -c ../hello-mingw
```

After adding a new (or upgrading/downgrading existing) dependency, it's a good idea to *deep-test* our project: run not only our own tests but also of its immediate (`--immediate|-i`) or even all (`--recursive|-r`) dependencies. For example:

```
$ bdep test -ai
in configuration @gcc:
test hello/test{testscript} ../hello-gcc/hello/hello/exe{hello}
test ../hello-gcc/libhello-1.0.0/tests/basics/exe{driver}

in configuration @clang:
test hello/test{testscript} ../hello-clang/hello/hello/exe{hello}
test ../hello-clang/libhello-1.0.0/tests/basics/exe{driver}
```

To get rid of a dependency, we simply remove it from the `manifest` file and synchronize the project. For example, assuming `libhello` is no longer mentioned as a dependency in our `manifests`:

```
$ bdep status
hello configured 0.1.0-a.0.19700101000000#1
      available  0.1.0-a.0.19700101000000#2

$ bdep sync
synchronizing:
  drop libhello/1.0.0 (unused)
  upgrade hello/0.1.0-a.0.19700101000000#2
```

If instead of building a dependency from source you would prefer to use a version that is installed by your system package manager, see Using System-Installed Dependencies. And for information on using dependencies that are not `build2` packages refer to Using Unpackaged Dependencies.

2.4 Upgrading and Downgrading Dependencies

Let's say we would like to try that `1.1.0` version we have seen in the `libhello` `git` repository. First, we need to add the repository to the `repositories.manifest` file:

```
role: prerequisite
location: https://git.build2.org/hello/libhello.git
```

Note that we don't need the `trust` value since `git` repositories are not authenticated.

To refresh the list of available dependency versions we use the **`bdep-fetch(1)`** command (or the `--fetch|-f` option to `status`):

```
$ bdep fetch
$ bdep status libhello
libhello configured 1.0.0 available [1.1.0]
```

To upgrade (or downgrade) dependencies we again use the **`bdep-sync(1)`** command. We can upgrade one or more specific dependencies by listing them as arguments to `sync`:

```
$ bdep sync libhello
synchronizing:
  new libformat/1.0.0 (required by libhello)
  new libprint/1.0.0 (required by libhello)
  upgrade libhello/1.1.0
  upgrade hello/0.1.0-a.0.19700101000000#3
```

Without an explicit version or the `--patch|-p` option, `sync` will upgrade the specified dependencies to the latest available versions. For example, if we don't like version `1.1.0`, we can downgrade it back to `1.0.0` by specifying the version explicitly (we pass `--old-available|-o` to `status` to see the old versions):

```
$ bdep status -o libhello
libhello configured 1.1.0 available (1.1.0) [1.0.0]

$ bdep sync libhello/1.0.0
synchronizing:
  drop libprint/1.0.0 (unused)
  drop libformat/1.0.0 (unused)
  downgrade libhello/1.0.0
  reconfigure hello/0.1.0-a.0.19700101000000#3
```

The available versions are listed in the descending order with `[]` indicating that the version is only available as a dependency and `()` marking the current version.

Instead of specific dependencies we can also upgrade (`--upgrade|-u`) or patch (`--patch|-p`) immediate (`--immediate|-i`) or all (`--recursive|-r`) dependencies of our project.

As a more realistic example, version `1.1.0` of `libhello` depends on two other libraries: `libformat` and `libprint`. Here is our project's dependency tree while we were still using that version:

```
$ bdep status -r
hello configured 0.1.0-a.0.19700101000000#3
  libhello ^1.0.0 configured 1.1.0
    libformat ^1.0.0 configured 1.0.0
    libprint ^1.0.0 configured 1.0.0
```

A typical conservative dependency management workflow would look like this:

```
$ bdep status -fi  # refresh and examine immediate dependencies
hello configured 0.1.0-a.0.19700101000000#3
  libhello configured 1.1.0 available [2.0.0] [1.2.0] [1.1.2] [1.1.1]

$ bdep sync -pi    # upgrade immediate to latest patch version
synchronizing:
  upgrade libhello/1.1.2
  reconfigure hello/0.1.0-a.0.19700101000000#3
continue? [Y/n] y
```

Notice that in case of such mass upgrades you are prompted for confirmation before anything is actually changed (unless you pass `--yes|-y`).

In contrast, the following would be a fairly aggressive workflow where we upgrade everything to the latest available version (version constraints permitting; here we assume `^1.0.0` was used for all the dependencies):

```
$ bdep status -fr  # refresh and examine all dependencies
hello configured 0.1.0-a.0.19700101000000#3
  libhello configured 1.1.0 available [2.0.0] [1.2.0] [1.1.1]
    libprint configured 1.0.0 available [2.0.0] [1.1.0] [1.0.1]
    libformat configured 1.0.0 available [2.0.0] [1.1.0] [1.0.1]
```

```
$ bdep sync -ur    # upgrade all to latest available version
synchronizing:
  upgrade libprint/1.1.0
  upgrade libformat/1.1.0
  upgrade libhello/1.2.0
  reconfigure hello/0.1.0-a.0.19700101000000#3
continue? [Y/n] y
```

We can also have something in between: patch all (`sync -pr`), upgrade immediate (`sync -ui`), or even upgrade immediate and patch the rest (`sync -ui` followed by `sync -pr`).

2.5 Versioning and Release Management

Let's now discuss versioning and release management and, yes, that strange-looking `0.1.0-a.0.19700101000000` we keep seeing. While a build system project doesn't need a version and a `bpkg` package can use custom versioning schemes (see Package Version), a project managed by `bdep` must use *standard versioning*. A dependency, which is a `bpkg` package, need not use standard versioning.

Standard versioning (*stdver*) is a semantic versioning (*semver*) scheme with a more precisely defined pre-release component and without any build metadata.

If you believe that *semver* is just *`major.minor.patch`*, then in your worldview *stdver* would be the same as *semver*. In reality, *semver* also allows loosely defined pre-release and build metadata components. For example, `1.2.3-beta.1+build.23456` is a valid *semver*.

A standard version has the following form:

major.*minor*.*patch*[-*prerel*]

The *major*, *minor*, and *patch* components have the same meaning as in *semver*. The *prerel* component is used to provide *continuous versioning* of our project between releases. Specifically, during development of a new version we may want to publish several pre-releases, for example, alpha or beta. In between those we may also want to publish a number of snapshots, for example, for CI. With continuous versioning all these releases, pre-releases, and snapshots are assigned unique, properly ordered versions.

Continuous versioning is a cornerstone of the `build2` project dependency management. In case of snapshots, an appropriate version is assigned automatically in cooperation with your VCS.

The *prerel* component for a pre-release has the following form:

(**a**|**b**).*num*

Here **a** stands for alpha, **b** stands for beta, and *num* is the alpha/beta number. For example:

```
1.1.0        # final              release  for 1.1.0
1.2.0-a.1    # first  alpha   pre-release  for 1.2.0
1.2.0-a.2    # second alpha   pre-release  for 1.2.0
1.2.0-b.1    # first  beta    pre-release  for 1.2.0
1.2.0        # final              release  for 1.2.0
```

The *prerel* component for a snapshot has the following form:

(**a**|**b**).*num*.*snapsn*[.*snapid*]

Where *snapsn* is the snapshot sequence number and *snapid* is the snapshot id. In case of `git`, *snapsn* is the commit timestamp in the `YYYYMMDDhhmmss` form and UTC timezone while *snapid* is a 12-character abbreviated commit id. For example:

```
1.2.3-a.1.20180319215815.26efe301f4a7
```

Notice also that a snapshot version is ordered *after* the corresponding pre-release version. That is, `1.2.3-a.1 < 1.2.3-a.1.1`. As a result, it is customary to start the development of a new version with `X.Y.Z-a.0.z`, that is, a snapshot after the (non-existent) zero'th alpha release. We will explain the meaning of **z** in this version momentarily. The following chronologically-ordered versions illustrate a typical release flow of a project that uses `git` as its VCS:

```
0.1.0-a.0.19700101000000             # snapshot (no commits yet)
0.1.0-a.0.20180319215815.26efe301f4a7  # snapshot (first commit)
...                                    # more commits/snapshots
0.1.0-a.1                              # pre-release (first alpha)
0.1.0-a.1.20180319221826.a6f0f41205b8  # snapshot
...                                    # more commits/snapshots
0.1.0-a.2                              # pre-release (second alpha)
0.1.0-a.2.20180319231937.b701052316c9  # snapshot
...                                    # more commits/snapshots
0.1.0-b.1                              # pre-release (first beta)
0.1.0-b.1.20180319242038.c812163417da  # snapshot
...                                    # more commits/snapshots
0.1.0                                  # release
0.2.0-a.0.20180319252139.d923274528eb  # snapshot (first in 0.2.0)
...
```

For a more detailed discussion of standard versioning and its support in `build2` refer to Version Module.

Let's now see how this works in practice by publishing a couple of versions for our `hello` project. By now it should be clear what that `0.1.0-a.0.19700101000000` means – it is the first snapshot version of our project. Since there are no commits yet, it has the UNIX epoch as its commit timestamp. As the first step, let's try to commit our project and see what changes:

```
$ git add .
$ git commit -m "Start hello project"

$ bdep status
hello configured 0.1.0-a.0.19700101000000
      available  0.1.0-a.0.20180507062614.ee006880fc7e
```

Just like with changes to dependency information, `status` has detected that a new (snapshot) version of our project is available for synchronization.

Another way to view the project's version (which works even if we are not using `bdep`) is with the build system's `info` operation:

```
$ b info
project: hello
version: 0.1.0-a.0.20180507062614.ee006880fc7e
summary: hello executable project
...
```

Let's synchronize with the default build configuration:

```
$ bdep sync
synchronizing:
  upgrade hello/0.1.0-a.0.20180507062614.ee006880fc7e

$ bdep status
hello configured 0.1.0-a.0.20180507062614.ee006880fc7e
```

Notice that we didn't have to manually change the version anywhere. All we had to do was commit our changes and a new snapshot version was automatically derived by `build2` from the new `git` commit. Without this automation continuous versioning would hardly be practical.

If we now make another commit, we will see a similar picture:

```
$ bdep status
hello configured 0.1.0-a.0.20180507062614.ee006880fc7e
      available  0.1.0-a.0.20180507062615.8fb9de05b38f
```

Note that you don't need to manually run `sync` after every commit. As discussed earlier, you can simply run the build system to update your project and things will get automatically synchronized if necessary.

Ok, time for our first release. Let's start with `0.1.0-a.1`. Unlike snapshots, for pre-releases as well as final releases we have to update the version in the `manifest` file manually:

```
version: 0.1.0-a.1
```

The `manifest` file is the singular place where we specify the package version. The build system's `version` module makes it available in various forms in buildfiles and even source code.

To ensure continuous versioning, this change to version must be the last commit for this (pre-)release which itself must be immediately followed by a second change to the version starting the development of the next (pre-)release. We also recommend that you tag the release commit with a tag name in the **v**`X.Y.Z` form.

Having regular release tag names with the **v** prefix allows one to distinguish them from other tags, for example, with wildcard patterns.

Here is the release workflow for our example:

```
$ git commit -a -m "Release version 0.1.0-a.1"
$ git tag -a v0.1.0-a.1 -m "Tag version 0.1.0-a.1"
$ git push --follow-tags

# Version 0.1.0-a.1 is now public.

$ edit manifest  # change 'version: 0.1.0-a.1.z'
$ git commit -a -m "Change version to 0.1.0-a.1.z"
$ git push

# Master is now open for business.
```

In the future release management will be automated with a `bdep-release(1)` command.

Notice also that when specifying a snapshot version in `manifest` we use the special **z** snapshot value (for example, `0.1.0-a.1.z`) which is recognized and automatically replaced by `build2` with, in case of `git`, a commit timestamp and id (refer to Version Module for details).

Publishing the final release is exactly the same. For completeness, here are the commands:

```
$ edit manifest  # change 'version: 0.1.0'
$ git commit -a -m "Release version 0.1.0"
$ git tag -a v0.1.0 -m "Tag version 0.1.0"
$ git push --follow-tags

$ edit manifest  # change 'version: 0.2.0-a.0.z'
$ git commit -a -m "Change version to 0.2.0-a.0.z"
$ git push
```

One sticky point of continuous versioning is choosing the next version. For example, above should we continue with `0.1.1-a.0`, `0.2.0-a.0`, or `1.0.0-a.0`? The important rule to keep in mind is that we can jump forward to any further version at any time and without breaking continuous versioning. But we can never jump backwards.

For example, we can start with `0.2.0-a.0` but if we later realize that this will actually be a new major release, we can easily change it to `1.0.0-a.0`. As a result, the general recommendation is to start conservatively by either incrementing the patch or the minor version component. The recommended strategy is to increment the minor component and, if required, release patch versions from a separate branch (created by branching off from the release commit).

Note also that you don't have to make any pre-releases if you don't need them. While during development you would still keep the version as `X.Y.Z-a.0`, at release you simply change it directly to the final `X.Y.Z`.

When publishing the final release you may also want to clean up now obsolete pre-release tags. For example:

```
$ git tag -l 'v0.1.0-*' | xargs git push --delete origin
$ git tag -l 'v0.1.0-*' | xargs git tag --delete
```

While at first removing such tags may seem like a bad idea, pre-releases are by nature temporary and their use only makes sense until the final release is published.

Also note that having a `git` repository with a large number of published but unused version tags may result in a significant download overhead.

Let's also briefly discuss in which situations we should increment each of the version components. While *semver* gives basic guidelines, there are several ways to apply them in the context of C/C++ where there is a distinction between binary and source compatibility. We recommend that you reserve *patch* releases for specific bug fixes and security issues that you can guarantee with a high level of certainty to be binary-compatible. Otherwise, if the changes are source-compatible, increment *minor*. And if they are breaking (that is, the user code likely will need adjustments), increment *major*. During early development, when breaking changes are frequent, it is customary to use the `0.Y.Z` versions where `Y` effectively becomes the *major* component. Again, refer to the Version Module for a more detailed discussion of this topic.

2.6 Developing Multiple Packages and Projects

How does a library like `libhello` get developed? It's possible someone woke up one day and realized that they were going to build a useful library that everyone was going to use. But somehow this doesn't feel like how it really works. In the real world things start organically: someone had a project like `hello` and then needed the same functionality in another project. Or someone else needed it and asked the author to factor it out into a library. For this approach to work, however, moving such common functionality into a library and then continue its parallel development must be a simple, frictionless process. Let's see how this works in `build2`.

First, we need to decide whether to make `libhello` another package in our `hello` project (that is, in the same `git` repository) or a separate project (with a separate repository). Both arrangements are equally well supported.

A multi-package project works best if all the packages have the same version and are released together. While the packages themselves can have different versions (since each has its own `manifest`), in this scenario following the release tagging recommendations discussed earlier will be problematic.

Let's start with a separate project since it is simpler. As the first step we use **bdep-new(1)** to create a new library project next to our `hello`:

```
$ bdep new -t lib -l c++ libhello
created new library project libhello in /tmp/libhello/

$ ls
hello/
libhello/
hello-gcc/
hello-clang/
```

Our two projects will be sharing the same set of build configurations, so next we initialize `libhello` in `hello-gcc` and `hello-clang`:

```
$ cd libhello

$ bdep init -A ../hello-gcc @gcc
initializing in project /tmp/libhello/
added configuration @gcc /tmp/hello-gcc/ default,auto-synchronized
synchronizing:
  new libhello/0.1.0-a.0.19700101000000

$ bdep init -A ../hello-clang @clang
initializing in project /tmp/libhello/
added configuration @clang /tmp/hello-clang/ auto-synchronized
synchronizing:
  new libhello/0.1.0-a.0.19700101000000
```

If two or more projects share the same build configuration, then all of them are always synchronized at once, regardless of the originating project. It also makes sense to have the same default configuration and use identical configuration names in all the projects.

The last step is to move the desired functionality from `hello` to `libhello` and at the same add a dependency on `libhello`, just as we did earlier (add a `depends` entry to `manifest`, then import the library in `buildfile`, and so on). One interesting question is what to put as a prerequisite repository in `repositories.manifest`. Our own setup will work even if we don't put anything there – the dependency will be automatically resolved to our local version of `libhello` since we have initialized it in all our build configurations. However, in case our `hello` repository is used by someone else, it's a good idea to add the remote `git` repository for `libhello` as a prerequisite.

By now you have probably realized that our project directory is just another type of package repository. See **bpkg-repository-types(1)** for more information.

And that's it, now we can build and test our new arrangement:

```
$ cd ../hello   # back to hello project root
$ bdep test -i
c++ ../libhello/libhello/cxx{hello}
c++ ../libhello/tests/basics/cxx{driver}
c++ hello/cxx{hello}
ld ../hello-gcc/libhello/libhello/libs{hello}
ld ../hello-gcc/libhello/tests/basics/exe{driver}
ld ../hello-gcc/hello/hello/exe{hello}
test ../hello-gcc/libhello/tests/basics/exe{driver}
test hello/test{testscript} ../hello-gcc/hello/hello/exe{hello}
```

This is also the approach we would use if we wanted to fix a bug in someone else's library. That is, we would clone their project repository and initialize it in the build configurations of our project which will "upgrade" the dependency to use the local version. Then we make the fix, submit it upstream, and continue using the local version until our fix is merged/published, at which point we deinitialize the project and switch back to using the upstream version.

Let's now examine the second option: making `libhello` a package inside `hello`. Here is the original structure of our `hello` project:

```
hello/
âââ .git/
âââ build/
âââ hello/
âÂ Â  âââ hello.cxx
âÂ Â  âââ buildfile
âââ buildfile
âââ manifest
âââ repositories.manifest
```

As the first step, we move the `hello` program into its own subdirectory:

```
hello/
âââ .git/
âââ hello/
âÂ Â  âââ build/
âÂ Â  âââ hello/
âÂ Â  âÂ Â  âââ hello.cxx
âÂ Â  âÂ Â  âââ buildfile
âÂ Â  âââ buildfile
âÂ Â  âââ manifest
âââ repositories.manifest
```

Next we again use **`bdep-new(1)`** to create a new library but this time as a package inside an already existing project:

```
$ cd hello
$ bdep new --package -t lib -l c++ libhello
created new library package libhello in /tmp/hello/libhello/
```

Let's see what our project looks like now:

```
hello/
âââ .git/
âââ hello/
âÂ Â  âââ ...
âÂ Â  âââ manifest
âââ libhello/
âÂ Â  âââ ...
âÂ Â  âââ manifest
âââ packages.manifest
âââ repositories.manifest
```

Notice that, as discussed earlier, `repositories.manifest` belongs to the project (repository) while `manifest` – to the package.

Besides the `libhello` directory the `new` command also created the `packages.manifest` file in the root directory of our project. Let's take a look inside:

```
$ cat packages.manifest
: 1
location: libhello/
```

Up until now our `hello` was a simple, single-package project that didn't need this file – `manifest` in its root directory was sufficient (see **`bpkg-repository-types(1)`** for details on the project repository structure). But now it contains several packages and we need to specify where they are located within the project. So let's go ahead and add the location of the `hello` package:

```
$ cat packages.manifest
: 1
location: libhello/
:
location: hello/
```

Packages in a project can reside next to each other or in subdirectories but they cannot nest. When published to an archive-based repository, each such package will be placed into its own archive.

Next we initialize the new package in all our build configurations:

```
$ cd libhello
$ bdep init -a
initializing in project /tmp/hello/
in configuration @gcc:
synchronizing:
  upgrade hello/0.1.0-a.0.19700101000000#1
  new libhello/0.1.0-a.0.19700101000000
```

```
in configuration @clang:
synchronizing:
  upgrade hello/0.1.0-a.0.19700101000000#1
  new libhello/0.1.0-a.0.19700101000000
```

Notice that the `hello` package has been "upgraded" to reflect its new location.

Finally, as before, we move the desired functionality from `hello` to `libhello` and at the same time add a dependency on `libhello`. Note, however, that in this case we don't need to add anything to `repositories.manifest` since both packages are in the same project (repository). And that's it, now we can build and test our new arrangement:

```
$ cd ..      # back to hello project root
$ bdep test
c++ libhello/libhello/cxx{hello}
c++ libhello/tests/basics/cxx{driver}
c++ hello/hello/cxx{hello}
ld ../hello-gcc/libhello/libhello/libs{hello}
ld ../hello-gcc/libhello/tests/basics/exe{driver}
ld ../hello-gcc/hello/hello/exe{hello}
test ../hello-gcc/libhello/tests/basics/exe{driver}
test hello/hello/test{testscript} ../hello-gcc/hello/hello/exe{hello}
```

2.7 Package Consumption

Ok, now that we have published a few releases of `hello`, how would the users of our project get them? While they could clone the repository and use `bdep` just like we did, this is more of a development rather than consumption workflow. For consumption it is much easier to use the package dependency manager, **`bpkg(1)`**, directly.

Note that this approach also works for libraries in case you wish to use them in a project with a build system other than `build2`. See Using Unpackaged Dependencies for background on cross-build system library consumption.

First, we create a suitable build configuration with the **`bpkg-cfg-create(1)`** command. We can use the same place for building all our tools so let's call the directory `tools`. Seeing that we are only interested in using (rather than developing) such tools, let's build them optimized and also configure a suitable installation location:

```
$ bpkg create -d tools cc        \
  config.cxx=g++                 \
  config.cc.coptions=-O3         \
  config.install.root=/usr/local \
  config.install.sudo=sudo
created new configuration in /tmp/tools/

$ cd tools
```

The same step on Windows using Visual Studio would look like this (again, remember to run this from the Visual Studio development command prompt):

```
$ bpkg create -d tools cc ^
  config.cxx=cl          ^
  config.cc.coptions=/O2  ^
  config.install.root= C:\install
```

To fetch and build packages (as well as all their dependencies) we use the **bpkg-pkg-build(1)** command. We can use either an archive-based repository like cppget.org or build directly from `git`:

```
$ bpkg build hello@https://git.build2.org/hello/hello.git
fetching from https://git.build2.org/hello/hello.git
  new libformat/1.0.0 (required by libhello)
  new libprint/1.0.0 (required by libhello)
  new libhello/1.1.0 (required by hello)
  new hello/1.0.0
continue? [Y/n] y
configured libformat/1.0.0
configured libprint/1.0.0
configured libhello/1.1.0
configured hello/1.0.0
c++ libprint-1.0.0/libprint/cxx{print}
c++ hello-1.0.0/hello/cxx{hello}
c++ libhello-1.1.0/libhello/cxx{hello}
c++ libformat-1.0.0/libformat/cxx{format}
ld libprint-1.0.0/libprint/libs{print}
ld libformat-1.0.0/libformat/libs{format}
ld libhello-1.1.0/libhello/libs{hello}
ld hello-1.0.0/hello/exe{hello}
updated hello/1.0.0
```

Passing a repository URL to the `build` command is a shortcut to the following sequence of commands:

```
$ bpkg add https://git.build2.org/hello/hello.git  # add repository
$ bpkg fetch                                       # fetch package list
$ bpkg build hello                                 # build package by name
```

Once built, we can install the package to the location that we have specified with `config.install.root` using the **bpkg-pkg-install(1)** command:

```
$ bpkg install hello
...
install libformat-1.0.0/libformat/libs{format}
install libprint-1.0.0/libprint/libs{print}
install libhello-1.1.0/libhello/libs{hello}
install hello-1.0.0/hello/exe{hello}

$ hello World
Hello, World!
```

If on your system the installed executables don't run from `/usr/local` because of the unresolved shared libraries (or if you are installing somewhere else, such as `/opt`), then the easiest way to fix this is with *rpath*. Simply add the following configuration variable when creating the build configuration (or as an argument to the `install` command):

```
config.bin.rpath=/usr/local/lib
```

Note to Windows users: this is not an issue on this platform since executables and shared (DLL) libraries are installed into the same subdirectory (`bin`) of the installation directory.

The installation contents and layout under `config.install.root` would be along these lines:

```
/usr/local/
âââ bin/
âÂ Â  âââ hello
âââ include/
âÂ Â  âââ libformat/
âÂ Â  âÂ Â  âââ export.hxx
âÂ Â  âÂ Â  âââ format.hxx
âÂ Â  âÂ Â  âââ version.hxx
âÂ Â  âââ libhello/
âÂ Â  âÂ Â  âââ export.hxx
âÂ Â  âÂ Â  âââ hello.hxx
âÂ Â  âÂ Â  âââ version.hxx
âÂ Â  âââ libprint/
âÂ Â      âââ export.hxx
âÂ Â      âââ print.hxx
âÂ Â      âââ version.hxx
âââ lib/
âÂ Â  âââ libformat-1.0.so
âÂ Â  âââ libformat.so -> libformat-1.0.so
âÂ Â  âââ libhello-1.1.so
âÂ Â  âââ libhello.so -> libhello-1.1.so
âÂ Â  âââ libprint-1.0.so
âÂ Â  âââ libprint.so -> libprint-1.0.so
âÂ Â  âââ pkgconfig
âÂ Â      âââ libformat.shared.pc
âÂ Â      âââ libhello.shared.pc
âÂ Â      âââ libprint.shared.pc
âââ share/
    âââ doc/
        âââ libformat/
        âÂ Â  âââ manifest
        âââ libhello/
        âÂ Â  âââ manifest
        âââ libprint/
            âââ manifest
```

The installation locations of various types of files (executables, libraries, headers, documentation, etc.) can be customized using a number of the `config.install.*` variables with the most commonly used ones and their defaults (relative to `config.install.root`) listed below (see the `install` build system module documentation for the complete list).

```
config.install.bin     = root/bin/
config.install.lib     = root/lib/
config.install.doc     = root/share/doc/
config.install.man     = root/share/man/
config.install.include = root/include/
```

If we need to uninstall a previously installed package, there is the **bpkg-pkg-uninstall(1)** command:

```
$ bpkg uninstall hello
uninstall hello-1.0.0/hello/exe{hello}
uninstall libhello-1.1.0/libhello/libs{hello}
uninstall libprint-1.0.0/libprint/libs{print}
uninstall libformat-1.0.0/libformat/libs{format}
...
```

To upgrade or downgrade packages we again use the `build` command. Here is a typical upgrade workflow:

```
$ bpkg fetch              # refresh available package list
$ bpkg status             # see if new versions are available

$ bpkg uninstall hello    # uninstall old version
$ bpkg build     hello    # upgrade to the latest version
$ bpkg install   hello    # install new version
```

Similar to `bdep`, to downgrade we have to specify the desired version explicitly. There are also the `--upgrade|-u` and `--patch|-p` as well as `--immediate|-i` and `--recursive|-r` options that allow us to upgrade or patch packages that we have built and/or their immediate or all dependencies (see **bpkg-pkg-build(1)** for details). For example, to make sure everything is patched, run:

```
$ bpkg fetch
$ bpkg build -pr
```

If a package is no longer needed, we can remove it from the configuration with **bpkg-pkg-drop(1)**:

```
$ bpkg drop hello
following dependencies were automatically built but
will no longer be used:
  libhello
  libformat
  libprint
drop unused packages? [Y/n] y
  drop hello
  drop libhello
  drop libformat
  drop libprint
continue? [Y/n] y
```

```
purged hello
purged libhello
purged libformat
purged libprint
```

2.8 Using System-Installed Dependencies

Our operating system might already have a package manager (which we will refer to as *system package manager*) and for various reasons we may want to use the system-installed version of a dependency rather than building one from source.

Using system-installed versions works best for mature rather than rapidly-developed packages since for the latter you often need to track the latest version (which may not yet be available from the system repository) and/or test with multiple versions (which is not something that many system package managers support).

We can instruct `build2` to configure a dependency package as available from the system rather than building it from source. Let's see how this works in an example. Say, we want to use `libsqlite3` in our `hello` project.

The first step is to add it as a dependency, just like we did for `libhello`. That is, add another `depends` entry to `manifest`, then import it in `buildfile`, and so on.

Note that the dependency still has to be packaged and available from one of the project's prerequisite repositories. However, it can be a *stub* – a package that does not contain any source code and that can only be "obtained" from the system (see Package Version for details). See also Using Unpackaged Dependencies for how to deal with dependencies that are not packaged.

Now, if we just run `sync` or try to build our project, `build2` will download and build the new dependency from source, just like it did for `libhello`. Instead, we can issue an explicit `sync` command that configures the `libsqlite3` package as coming from the system:

```
$ bdep sync ?sys:libsqlite3
synchronizing:
  configure sys:libsqlite3/*
  upgrade hello/0.1.0-a.0.19700101000000#3
```

Here **`?`** is a package *flag* that instructs `build2` to treat it as a dependency and **`sys`** is a package *scheme* that tells `build2` it comes from the system. See **`bpkg-pkg-build(1)`** for details.

We can have some build configurations using a system-installed version of a dependency while others building it from source, for example, for testing.

The system-installed dependency doesn't really have to come from the system package manager. It can also be manually installed and, as discussed in Using Unpackaged Dependencies, not necessarily into the system-default location like `/usr/local`.

Currently, unless we specify the installed version explicitly, a system-installed package is assumed to satisfy any dependency constraint. In the future, build2 will automatically query commonly used system package managers for the installed version and maybe even request installation of the absent packages. To support this functionality, the package manifest may need to specify package name mappings for various system package managers (which is the rationale behind stub packages).

2.9 Using Unpackaged Dependencies

Generally, we will have a much better time if all our dependencies come as build2 packages. Unfortunately, this won't always be the case in the real world and some libraries that you may need will use other build systems.

There is also the opposite problem: you may want to consume a library that uses build2 in a project that uses a different build system. For that refer to Package Consumption.

The standard way to consume such unpackaged libraries is to install them (not necessarily into a system-default location like /usr/local) so that we have a single directory with their headers and a single directory with their libraries. We can then configure our builds to use these directories when searching for imported libraries.

Needless to say, none of the build2 dependency management mechanisms such as version constraints or upgrade/downgrade will work on such unpackaged libraries. You will have to manage all these yourself manually.

Let's see how this all works in an example. Say, we want to use libextra that uses a different build system in our hello project. The first step is to manually build and install this library for each build configuration that we have. For example, we can install all such unpackaged libraries into unpkg-gcc and unpkg-clang, next to our hello-gcc and hello-clang build configurations:

```
$ ls
hello/
hello-gcc/
unpkg-gcc/
hello-clang/
unpkg-clang/
```

If you would like to try this out but don't have a suitable libextra, you can create and install one with these commands:

```
$ bdep new -t lib -l c++ libextra -C libextra-gcc cc config.cxx=g++
$ b install: libextra-gcc/ config.install.root=/tmp/unpkg-gcc
```

If we look inside one of these `unpkg-*` directories, we should see something like this:

```
$ tree unpkg-gcc
unpkg-gcc
âââ include
âÂ Â  âââ libextra
âÂ Â      âââ extra.hxx
âââ lib
    âââ libextra.a
    âââ libextra.so
    âââ pkgconfig
        âââ libextra.pc
```

Notice that `libextra.pc` – it's a **`pkg-config(1)`** file that contains any extra compile and link options that may be necessary to consume this library. This is the *de facto* standard for build systems to communicate library build information to each other and is today supported by most commonly used implementations. Speaking of `build2`, it both recognizes `.pc` files when consuming third-party libraries and automatically produces them when installing its own.

While this may all seem foreign to Windows users, there is nothing platform-specific about this approach, including support for `pkg-config`, which, at least in case of `build2`, works equally well on Windows.

Next, we create a build configuration and configure it to use one of these `unpkg-*` directories (replace `...` with the absolute path):

```
$ bdep init -C ../hello-gcc @gcc cc config.cxx=g++ \
  config.cc.poptions=-I.../unpkg-gcc/include       \
  config.cc.loptions=-L.../unpkg-gcc/lib
```

If using Visual Studio, replace `-I` with `/I` and `-L` with `/LIBPATH:`.

Alternatively, if you want to reconfigure one of the existing build configurations, then simply edit the `build/config.build` file (that is, `hello-gcc/build/config.build` in our case) and adjust the `poptions` and `loptions` values. Or you can use the build system directly to reconfigure the build configuration (see **`b(1)`** for details):

```
b configure: ../hello-gcc/                      \
  config.cc.poptions+=-I.../unpkg-gcc/include \
  config.cc.loptions+=-L.../unpkg-gcc/lib
```

If all the unpackaged libraries included `.pc` files, then the `-L` alone would have been sufficient. However, it doesn't hurt to also add `-I`, for good measure.

Once this is done, adjust your `buildfile` to import the library:

```
import libs += libextra%lib{extra}
```

And your source code to use it:

```
#include <libextra/extra.hxx>
```

Notice that we don't add the corresponding `depends` value to the project's `manifest` since this library is not a package. However, it is a good idea to instead add a `requires` entry as a documentation to users of our project.