The build2 Packaging Guide

Revision 0.17, June 2024
This revision of the document describes the build2 toolchain 0.17.x series.

Table of Contents

Preface

This document provides guidelines for converting third-party C/C++ projects to the `build2` build system and making them available as packages from cppget.org, the `build2` community's central package repository. For additional information, including documentation for individual `build2` toolchain components, man pages, HOWTOs, etc., refer to the project Documentation page.

1 Introduction

The aim of this guide is to ease the conversion of third-party C/C++ projects to the `build2` build system and publishing them to the cppget.org package repository by codifying the best practices and techniques. By following the presented guidelines you will also make it easier for others to review your work and help with ongoing maintenance.

A `build2`-based project can only consume packages that use the `build2` build system (with the exception of system-installed packages). In other words, there is no support for "wrapping" or otherwise adapting third-party projects' existing build systems. While replacing the build system unquestionably requires more work upfront, the `build2` project's experience is that the long-term benefits of this effort are well justified (see How does `build2` compare to other package managers? for details).

The primary focus of this guide is existing C/C++ projects that use a different build system and that are maintained by a third-party, which we will refer to as *upstream*. Unless upstream is willing to incorporate support for `build2` directly into their repository, such projects are normally packaged for `build2` in a separate `git` repository under the github.com/build2-packaging organization. Note, however, that many of the presented guidelines are also applicable when converting your own projects (that is, where you are the upstream) as well as projects that use languages other than C or C++.

Most C/C++ packages that are published to cppget.org are either libraries or executables (projects that provide both are normally split into several packages) with libraries being in the strong majority. Libraries are also generally more difficult to build correctly. As a result, this guide uses libraries as a baseline. In most cases, a library-specific step is easily distinguished as such and can be skipped when dealing with executables. And in cases where a more nuanced change is required, a note will be provided.

At the high-level, packaging a third-party project involves the following steps:

1. Create the `git` repository and import upstream source code.
2. Generate `buildfile` templates that match upstream layout.
3. Tweak the generated `buildfiles` to match upstream build.
4. Test locally and using the `build2` CI service.
5. Release and publish the package to cppget.org.

Once this process is completed and the package is published, new releases normally require a small amount of work provided there are no drastic changes in the upstream layout or build. The sequence of steps for a new release would typical look like this:

1. Add new and/or remove old upstream source code, if any.
2. Tweak `buildfiles` to match changes to upstream build, if any.
3. Test locally and using the `build2` CI service.
4. Release and publish the package to cppget.org.

While packaging a simple library or executable is relatively straightforward, the C and C++ languages and their ecosystems are infamous for a large amount of variability in the platforms, compilers, source code layouts, and build systems used. This leads to what looks like an endless list of special considerations that are only applicable in certain, more complex cases.

As result, the presented guidelines are divided into four chapters: Common Guidelines cover steps that are applicable to most packaging efforts. As mentioned earlier, these steps will assume packaging a library but they should be easy to adapt to executables. This chapter is followed by What Not to Do which covers the common packaging mistakes and omissions. These are unfortunately relatively common because experience with other build systems often does not translate directly to `build2` and some techniques (such as header-only libraries) are discouraged. The last two chapters are HOWTO and FAQ. The former covers the above-mentioned long list of special considerations that are only applicable in certain cases while the latter answer frequent packaging-related questions.

Besides the presented guidelines, you may also find the existing packages found in github.com/build2-packaging a good source of example material. The repositories pinned to the front page are the recommended starting point.

This guide assumes familiarity with the `build2` toolchain. At the minimum you should have read through The `build2` Toolchain Introduction and the Introduction chapter in the build system manual. Ideally, you should also have some experience using `build2` in your own projects.

In this guide we will only show the UNIX version of the commands. In most cases making a Windows version is a simple matter of adjusting paths and, if used, line continuations. And where this is not the case a note will be provided showing the equivalent Windows command.

1.1 Terminology

We use the term *upstream* to refer collectively to the third-party project as well as to its authors. For example, we may say, "upstream does not use semver" meaning that the upstream project does not use semver for versioning. Or we may say, "upstream released a new version" meaning that the upstream project's authors released a new version.

We will often use *upstream* as a qualifier to refer to a specific part of the upstream project. Commonly used qualified terms like this include:

upstream repository
: The version control (normally `git`) repository of the third-party project.

upstream source code
: The C/C++ source code that constitutes the third-party project.

upstream layout
: The directory structure and location of source code in the third-party project.

upstream build system
: The equivalents of `buildfiles` that are used by the third-party project to build its source code, run tests, etc. For example, if upstream uses CMake, then all the `CMakeLists.txt`, `*.cmake`, etc., files will constitute its build system.

To avoid confusion, in this guide we will always use the term *project* to refer to upstream and *package* to refer to its `build2` conversion, even though we would normally call our own `build2`-based work a project, not a package (see Project Structure for details on the `build2` terminology in this area). Some commonly used `build2`-side terms in this guide include:

package `git` repository
: The `git` repository that hosts the package of the upstream project.

multi-package repository
: Sometimes it makes sense to split the upstream project into multiple `build2` packages (for example, a library and an executable). In this case the package repository structure must become multi-package.

2 Common Guidelines

This chapter describes the recommended sequence of steps for packaging a third-party project for `build2` with the end-goal of publishing it to the cppget.org package repository.

2.1 Setup the package repository

This section covers the creation of the package `git` repository and the importation of the upstream source code.

2.1.1 Check if package repository already exists

Before deciding to package a third-party project you have presumably checked on cppget.org if someone has already packaged it. There are several other places that make sense to check as well:

- queue.cppget.org contains packages that have been submitted but not yet published.
- queue.stage.build2.org contains packages that have been submitted but can only be published after the next release of the `build2` toolchain (see Where to publish if

package requires staged toolchain? for background).

- github.com/build2-packaging contains all the third-party package repositories. Someone could already be working on the package but haven't yet published it.
- github.com/build2-packaging/WISHLIST contains as issues projects that people wish were packaged. These may contain offers to collaborate or announcements of ongoing work.

In all these cases you should be able to locate the package `git` repository and/or connect with others in order to collaborate on the packaging work. If the existing effort looks abandoned (for example, there hasn't been any progress for a while and the existing maintainer doesn't respond) and you would like to take over the package, get in touch.

2.1.2 Use upstream repository name as package repository name

It is almost always best to use the upstream repository name as the package repository name. If there is no upstream repository (for example, because the project doesn't use a version control system), the name used in the source archive distribution would be the natural fallback choice.

See Decide on the package name for the complete picture on choosing names.

2.1.3 Create package repository in personal workspace

For a third-party project, the end result that we are aiming for is a package repository under the github.com/build2-packaging organization.

We require all the third-party projects that are published to cppget.org to be under the github.com/build2-packaging organization in order to ensure some continuity in case the original maintainer loses interest, etc. You will still be the owner of the repository and by hosting your packaging efforts under this organization (as opposed to, say, your personal workspace) you make it easier for others to discover your work and to contribute to the package maintenance.

Note that this requirement does not apply to your own projects (that is, where you are the upstream) and where the `build2` support is normally part of the upstream repository.

Finally, a note on the use of `git` and GitHub: if for some reason you are unable to use either, get in touch to discuss alternatives.

However, the recommended approach is to start with a repository in your personal workspace and then, when it is ready or in a reasonably complete state, transfer it to github.com/build2-packaging. This gives you the freedom to make destructive changes to the repository (including deleting it and starting over) during the initial packaging work. It also removes the pressure to perform: you can give it a try and if things turn out more difficult than you expected, you can just drop the repository.

For repositories under github.com/build2-packaging the `master`/`main` branch is protected: it cannot be deleted and its commit history cannot be overwritten with a forced push.

While you can use any name for a repository under the personal workspace, under github.com/build2-packaging it should follow the Use upstream repository name as package repository name guideline. In particular, there should be no prefixes like `build2-` or suffixes like `-package`. If the repository under your personal workspace does not follow this guideline, you will need to rename it before transferring it to the github.com/build2-packaging organization.

There is one potential problem with this approach: it is possible that several people will start working on the same third-party project without being aware of each other's efforts. If the project you are packaging is relatively small and you don't expect it to take more than a day or two, then this is probably not worth worrying about. For bigger projects, however, it makes sense to announce your work by creating (or updating) the corresponding issue in github.com/build2-packaging/WISHLIST.

To put it all together, the recommended sequence of actions for this step:

1. Create a new empty repository under your personal workspace from the GitHub UI.
2. Set the repository description to `build2 package for <name>`, where `<name>` is the third-party project name.
3. Make the repository public (otherwise you won't be able to CI it).
4. Don't automatically add any files (`README`, `LICENSE`, etc).
5. Clone the empty repository to your machine (using the SSH protocol).

Since this is your personal repository, you can do the initial work directly in `master`/`main` or in a separate branch, it's up to you.

As a running example, let's assume we want to package a library called `foo` whose upstream repository is at `https://github.com/<upstream>/foo.git`. We have created its package repository at `https://github.com/<personal>/foo.git` (with the `build2 package for foo` description) and can now clone it:

```
$ git clone git@github.com:<personal>/foo.git
```

2.1.4 Initialize package repository with `bdep new`

Change to the root directory of the package repository that you have cloned in the previous step and run (continuing with our `foo` example):

```
$ cd foo/ # Change to the package repository root.
$ bdep new --type empty,third-party
$ tree -a .
./
|-- .bdep/
|Â Â  `-- ...
|-- .git/
|Â Â  `-- ...
|-- .gitattributes
|-- .gitignore
|-- README.md
`-- repositories.manifest
```

We use the special `third-party` sub-option which is meant for converting third-party projects to `build2`. See **`bdep-new(1)`** for details.

This command creates a number of files in the root of the repository:

`README.md`
: This is the repository `README.md`. We will discuss the recommended content for this file later.

`repositories.manifest`
: This file specifies the repositories from which this project will obtain its dependencies (see Adding and Removing Dependencies). If the project you are packaging has no dependencies, then you can safely remove this file (it's easy to add later if this changes). And for projects that do have dependencies we will discuss the appropriate changes to this file later.

`.gitattributes` and `.gitignore`
: These are the `git` infrastructure files for the repository. You shouldn't normally need to change anything in them at this stage (see the comments inside for details).

Next add and commit these files:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Initialize package repository"
```

In these guidelines we will be using the package repository setup that is capable of having multiple packages (referred to as *multi-package repository*). This is recommended even for upstream projects that only provides a single package because it gives us the flexibility of adding new packages at a later stage without having to perform a major restructuring of our repository.

Note also that upstream providing multiple packages is not the only reason we may end up having multiple `build2` packages. Another common reason is factoring tests into a separate package due to a dependency on a testing framework (see How do I handle tests that have extra dependencies? for background and details). While upstream adding new packages may not be very common, upstream deciding to use a testing framework is a lot more plausible.

The only notable drawback of using a multi-package setup with a single package is the extra subdirectory for the package and a few extra files (such as `packages.manifest` that lists the packages) in the root of the repository. If you are certain that the project that you are converting is unlikely to have multiple packages (for example, because you are the upstream) and won't need extra dependencies for its tests (a reasonable assumption for a C project), then you could instead go with the single-package repository where the repository root is the package root. See **`bdep-new(1)`** for details on how to initialize such a repository. In this guide, however, we will continue to assume a multi-package repository setup.

2.1.5 Add upstream repository as `git` submodule

If the third-party project is available from a `git` repository, then the recommended approach is to use the `git` submodule mechanism to make the upstream source code available inside the package repository, customarily in a subdirectory called `upstream/`.

While `git` submodules receive much criticism, in our case we use them exactly as intended: to select and track specific (release) commits of an external project. As a result, there is nothing tricky about their use for our purpose and all the relevant commands will be provided and explained, in case you are not familiar with this `git` mechanism.

Given the upstream repository URL, to add it as a submodule, run the following command from the package repository root (continuing with our `foo` example):

```
$ cd foo/ # Change to the package repository root.
$ git submodule add https://github.com/<upstream>/foo.git upstream
```

You should prefer `https://` over `git://` for the upstream repository URL since the `git://` protocol may not be accessible from all networks. Naturally, never use a URL that requires authentication, for example, SSH (SSH URLs start with `git@github.com` for GitHub).

Besides the repository URL, you also need the commit of the upstream release which you will be packaging. It is common practice to tag releases so the upstream tags would be the first place to check. Failing that, you can always use the commit id.

Assuming the upstream release tag you are interested in is called `vX.Y.Z`, to update the `upstream` submodule to point to this release commit, run the following commands:

```
$ cd upstream/
$ git checkout vX.Y.Z
$ cd ../
```

Then add and commit these changes:

```
$ git add .
$ git status
$ git commit -m "Add upstream submodule, vX.Y.Z"
```

Now we have all the upstream source code for the version that we are packaging available in the `upstream/` subdirectory of our repository.

The plan is to then use symbolic links (symlinks) to non-invasively overlay the `build2`-related files (`buildfile`, `manifest`, etc) with the upstream source code, if necessary adjusting upstream structure to split it into multiple packages and/or to better align with the source/output layouts recommended by `build2` (see Using Symlinks in `build2` Projects for background and rationale). But before we can start adding symlinks to the upstream source (and other files like `README`, `LICENSE`, etc), we need to generate the `buildfile` templates that match the upstream source code layout. This is the subject of the next section.

While on UNIX-like operating systems symlinks are in widespread use, on Windows it's a niche feature that unfortunately could be cumbersome to use (see Symlinks and Windows for details). However, the flexibility afforded by symlinks when packaging third-party projects is unmatched by any other mechanism and we therefore use them despite potentially sub-optimal packaging experience on Windows.

2.2 Create package and generate `buildfile` templates

This section covers the addition of the package to the repository we have prepared in the previous steps and the generation of the `buildfile` templates that match the upstream source code layout.

2.2.1 Decide on the package name

While choosing the package repository name was pretty straightforward, things get less clear cut when it comes to the package name.

If you need a refresher on the distinction between projects and packages, see Terminology.

Picking a name for a package that provides an executable is still relatively straightforward: you should use the upstream name (which is usually the same as the upstream project name) unless there is a good reason to deviate. One recommended place to check before deciding on a name is the Debian package repository. If their package name differs from upstream, then there is likely a good reason for that and it is worth trying to understand what it is.

Tip: when trying to find the corresponding Debian package, search for the executable file name in the package contents if you cannot find the package by its upstream name. Also consider searching in the `unstable` distribution in addition to `stable` for newer packages.

Picking a name for a package that provides a library is where things can get more complicated. While all the recommendations that have been listed for executables apply equally to libraries, there are additional considerations.

In build2 we recommend (but not require) that new library projects use a name that starts with lib in order to easily distinguish them from executables and avoid any clashes, potential in the future (see Canonical Project Structure for details). To illustrate the problem, consider the zstd project which provides a library and an executable. In upstream repository both are part of the same codebase that doesn't try to separate them into packages so that, for example, library could be used without downloading and building the executable. In build2, however, we do need to split them into two separate packages and both packages cannot be called zstd. So we call them zstd and libzstd.

If you are familiar with the Debian package naming policy, you will undoubtedly recognize this approach. In Debian all the library packages (with very few exceptions) start with the lib prefix. So when searching for an upstream name in the Debian package repository make sure to prefix it with lib (unless it already starts with this prefix, of course).

This brings the question of what to do about third-party libraries: should we add the lib prefix to the package name if it's not already there? Unfortunately, there is no clear cut answer and whichever decision you make, there will be drawbacks. Specifically, if you add the lib prefix, the main drawback is that the package name now deviates from the upstream name and if the project maintainer ever decides to add build2 support to the upstream repository, there could be substantial friction. On the other hand, if you don't add the lib prefix, then you will always run the risk of a future clash with an executable name. And, as was illustrated with the zstd example, a late addition of an executable won't necessarily cause any issues to upstream. As a result, we don't have a hard requirement for the lib prefix unless there is already an executable that would cause the clash (this applies even if it's not being packaged yet or is provided by an unrelated project). If you don't have a strong preference, we recommend that you add the lib prefix (unless it is already there). In particular, this will free you from having to check for any potential clashes. See How should I name packages when packaging third-party projects? for additional background and details.

To build some intuition for choosing package names, let's consider several real examples. We start with executables:

upstream project name	upstream executable name	Debian package name	build2 package repository name	build2 package name
byacc	byacc	byacc	byacc	byacc
sqlite	sqlite3	sqlite3	sqlite	sqlite3
vim	xxd	xxd	xxd	xxd
OpenBSD	m4	-	openbsd-m4	openbsd-m4
qtbase 5	moc	qtbase5-\ dev-tools	Qt5	Qt5Moc
qtbase 6	moc	qt6-base-\ dev-tools	Qt6	Qt6Moc

The examples are arranged from the most straightforward naming to the least. The last two examples show that sometimes, after carefully considering upstream naming, you nevertheless have no choice but to ignore it and forge your own path.

Next let’s look at library examples. Notice that some use the same `build2` package repository name as the executables above. This means they are part of the same multi-package repository.

upstream project name	upstream library name	Debian package name	build2 package repository name	build2 package name
libevent	libevent	libevent	libevent	libevent
brotli	brotli	libbrotli	brotli	libbrotli
zlib	zlib	zlib	zlib	libz
sqlite	libsqlite3	libsqlite3	sqlite	libsqlite3
libsig\ cplusplus	libsigc++	libsigc++	libsig\ cplusplus	libsigc++
qtbase 5	QtCore	qtbase5-dev	Qt5	libQt5Core
qtbase 6	QtCore	qt6-base-dev	Qt6	libQt6Core

If an upstream project is just a single library, then the project name is normally the same as the library name (but there are exceptions, like `libsigcplusplus` in the above table). However, when looking at the upstream repository that contains multiple components (libraries and/or executables, like `qtcore` in the above example), it may not be immediately obvious what the upstream’s library names are. In such cases, the corresponding Debian packages can really help clarify the situation. Failing that, look into the existing build system. In particular, if it generates the `pkg-config` file, then the name of this file is usually the upstream library name.

Looking at the names of the library binaries is less helpful because on UNIX-like systems they must start with the `lib` prefix. And on Windows the names of library binaries often embed extra information (static/import, debug/release, etc) and may not correspond directly to the library name.

And, speaking of multiple components, if you realize the upstream project provides multiple libraries and/or executables, then you need to decide whether to split them into separate `build2` packages and if so, how. Here, again, the corresponding Debian packages can be a good reference point. Note, however, that we often deviate from Debian’s splits, especially when it comes to libraries. Such differences are usually due to Debian focusing on binary packages while in `build2` we are focusing on source packages.

To give a few examples, `libevent` shown in the above table provides several libraries (`libevent-core`, `libevent-extra`, etc) and in Debian it is actually split into several binary packages along these lines. In `build2`, however, there is a single source package that provides all these libraries with everything except `libevent-core` being optional. An example which shows the decision made in a different direction would be the Boost libraries: in Debian all the header-only Boost libraries are bundled into a single package while in `build2` they are all separate packages.

The overall criteria here can be stated as follows: if a small family of libraries provide complimentary functionality (like `libevent`), then we put them all into a single package, usually making the additional functionality optional. However, if the libraries are independent (like Boost) or provide alternative rather than complimentary functionality (for example, like different backends in `imgui`), then we make them separate packages. Note that we never bundle an executable and a (public) library in a single package (because, when consumed, they usually require different dependency types: build-time vs run-time).

Note also that while it's a good idea to decide on the package split and all the package names upfront to avoid surprises later, you don't have to actually provide all the packages right away. For example, if upstream provides a library and an executable (like `zstd`), you can start with the library and the executable package can be added later (potentially by someone else).

In the "library and executable" case, if you plan to package both, the sensible strategy is to first completely package the library stopping short of releasing and publishing, then repeat the same process to package the executable, and finally release and publish both.

Admittedly, the recommendations in this section are all a bit fuzzy and one can choose different names or different package splits that could all seem reasonable. If you are unsure how to split the upstream project or what names to use, get in touch to discuss the alternatives. It can be quite painful to change these things after you have completed the remaining packaging steps.

Continuing with our `foo` example, we will follow the above recommendation and call the library package `libfoo`.

2.2.2 Decide on the package source code layout

Another aspect we need to decide on is the source code layout inside the package. Here we want to stay as close to the upstream layout as possible unless there are valid reasons to deviate. Staying close has the best chance of giving us a build without any compile errors since the header inclusion in the project can be sensitive to this layout. This also makes it easier for upstream to adopt the `build2` build.

Sometimes, however, there are good reasons for deviating from upstream, especially in cases where upstream is clearly following bad practices, for example including generically-named public headers without the library name as a subdirectory prefix. If you do decide to change the layout, it's usually less disruptive (to the build) to rearrange things at the outer levels than at the inner. For example, it should normally be possible to move/rename the top-level `tests/` directory or to place the library source files into a subdirectory.

Our overall plan is to create the initial layout and `buildfile` templates automatically using **`bdep-new(1)`** in the `--package` mode, then "fill" the package with upstream source code using symlinks, and finish off with tweaking the generated `buildfiles` to match the upstream build.

The main rationale for using **`bdep-new(1)`** instead of doing everything by hand is that there are many nuances in getting the build right and auto-generated `buildfiles` had years of refinement and fine-tuning. The familiar structure also makes it easier for others to understand your build, for example while reviewing your package submission or helping out with maintenance.

The **`bdep-new(1)`** command supports a wide variety of source layouts. While it may take a bit of time to understand the customization points necessary to achieve the desired layout for your first package, this experience will pay off in spades when you work on converting subsequent packages.

And so the focus of the following several steps is to iteratively discover the **`bdep-new(1)`** command line that best approximates the upstream layout. The recommended procedure is as follows:

1. Study the upstream source layout and existing build system.
2. Craft and execute the **`bdep-new(1)`** command line necessary to achieve the upstream layout.
3. Study the auto-generated `buildfiles` for things that don't fit and need to change. But don't rush to start manually editing the result. First get an overview of the required changes and then check if it's possible to achieve these changes automatically using one of **`bdep-new(1)`** sub-options. If that's the case, delete the package, and restart from step 2.

This and the following two sections discuss each of these steps in more detail and also look at some examples.

The first step above is to study the upstream project in order to understand where the various parts are (headers, sources, etc) and how they are built. Things that can help here include:

- Read through the existing build system definitions.
- Try to build the project using the existing build system.
- Try to install the project using the existing build system.
- Look into the Debian package contents to see if there are any differences with regards to the installation locations.

If while studying the upstream build system you notice other requirements, for example, the need to compile source files other than C/C++ (such as Objective-C/C++, assembler, etc) or the need to generate files from `.in` templates (or their `.cmake`/`.meson` equivalents), and are wondering how they would be handled in the `build2` build, see the Adjust source `buildfile`: extra requirements step for a collection of pointers.

For libraries, the first key pieces of information we need to find is how the public headers are included and where they are installed. The two common *good* practices is to either include the public headers with a library name as a subdirectory, for example, `#include <foo/util.h>`, or to include the library name into each public header name, for example, `#include <foo-util.h>` or `#include <foo.h>` (in the last example the header name is the library name itself, which is also fairly common). Unfortunately, there is also a fairly common *bad* practice: having generically named headers (such as `util.h`) included without the library name as a subdirectory.

The reason this is a bad practice is that libraries that have such headers cannot coexist, neither in the same build nor when installed. See How do I deal with bad header inclusion practice? if you encounter such a case. See Canonical Project Structure for additional background and details.

Where should we look to get this information? While the library source files sound like a natural place, oftentimes they include own headers with the `""` style inclusion, either because the headers are in the same directory or because the library build arranges for them to be found this way with additional header search paths. As a result, a better place to look could be the library’s examples and/or tests. Some libraries also describe which headers they provide and how to include them in their documentation.

The way public headers are included normally determines where they are installed. If they are included with a subdirectory, then they are normally installed into the same subdirectory in, say, `/usr/include/`. Continuing with the above example, a header that is included as `<foo/util.h>` would normally be installed as `/usr/include/foo/util.h`. On the other hand, if the library name is part of the header name, then the headers are usually (but not always) installed directly into, say, `/usr/include/`, for example as `/usr/include/foo-util.h`.

While these are the commonly used installation schemes, there are deviations. In particular, in both cases upstream may choose to add an additional subdirectory when installing (so the above examples will instead end up with, say, `/usr/include/foo-v1/foo/util.h` and `/usr/include/foo-v1/sub/foo-util.h`). See How do I handle extra header installation subdirectory? if you encounter such a case.

The inclusion scheme would normally also be recreated in the upstream source code layout. In particular, if upstream includes public headers with a subdirectory prefix, then this subdirectory would normally also be present in the upstream layout so that such a header can be included from the upstream codebase directly. As an example, let’s say we determined that public headers of `libfoo` are included with the `foo/` subdirectory, such as `<foo/util.hpp>`. One of the typical upstream layouts for such a library would look like this:

```
$ tree upstream/
upstream/
|-- include/
|Â Â  ·-- foo/
|Â Â      ·-- util.hpp
·-- src/
    |-- priv.hpp
    ·-- util.cpp
```

Notice how the `util.hpp` header is in the `foo/` subdirectory rather than in `include/` directly.

The second key piece of information we need to find is whether and, if so, how the public headers and sources are split. For instance, in the above example, we can see that public headers go into `include/` while sources and private headers go into `src/`. But they could also be combined in the same directory, for example, as in the following layout:

```
upstream/
`-- foo/
    |-- priv.hpp
    |-- util.cpp
    `-- util.hpp
```

In multi-package projects, for example, those that provide both a library and an executable, you would also want to understand how the sources are split between the packages.

If the headers and sources are split into different directories, then the source directory may or may not have the inclusion subdirectory, similar to the header directory. In the above split layout the `src/` directory doesn't contain the inclusion subdirectory (`foo/`) while the following layout does:

```
upstream/
|-- include/
|Â Â  `-- foo/
|Â Â      `-- util.hpp
`-- src/
    `-- foo/
        |-- priv.hpp
        `-- util.cpp
```

With the understanding of these key properties of upstream layout you should be in a good position to start crafting the **`bdep-new(1)`** command line that recreates it.

The `bdep-new` documentation uses slightly more general terminology compared to what we used in the previous section in order to also be applicable to projects that use modules instead of headers.

Specifically, the inclusion subdirectory (`foo/`) is called *source subdirectory* while the header directory (`include/`) and source directory (`src/`) are called *header prefix* and *source prefix*, respectively.

2.2.3 Craft `bdep new` command line to create package

The recommended procedure for this step is to read through the `bdep-new`'s SOURCE LAYOUT section (which contains a large number of examples) while experimenting with various options in an attempt to create the desired layout. If the layout you've got isn't quite right yet, simply remove the package directory along with the `packages.manifest` file and try again.

Next to `packages.manifest`, `bdep-new` will also create the "glue" `buildfile` that allows building all the packages from the repository root. You don't need to remove it when re-creating the package.

Let's illustrate this approach on the first split layout from the previous section:

```
upstream/
|-- include/
|Â Â  `-- foo/
|Â Â      `-- util.hpp
`-- src/
    |-- priv.hpp
    `-- util.cpp
```

We know it's split, so let's start with that and see what we get. Remember, our `foo` package repository that we have cloned and initialized earlier looks like this:

```
$ tree foo/
foo/
|-- upstream/
|-- .gitattributes
|-- .gitignore
|-- README.md
`-- repositories.manifest
```

Now we create the `libfoo` package inside:

```
$ cd foo/
$ bdep new --package --lang c++ --type lib,split libfoo
$ tree libfoo/
libfoo/
|-- include/
|Â Â  `-- libfoo/
|Â Â      `-- foo.hxx
`-- src/
    `-- libfoo/
        `-- foo.cxx
```

The outer structure looks right, but inside `include/` and `src/` things are a bit off. Specifically, the source subdirectory should be `foo/`, not `libfoo/`, there shouldn't be one inside `src/`, and the file extensions don't match upstream. All this can be easily tweaked, however:

```
$ rm -r libfoo/ packages.manifest
$ bdep new --package \
  --lang c++,cpp     \
  --type lib,split,subdir=foo,no-subdir-source \
  libfoo
$ tree libfoo/
libfoo/
|-- include/
|Â Â  `-- foo/
|Â Â      `-- foo.hpp
`-- src/
    `-- foo.cpp
```

The other `bdep-new` sub-options (see the **`bdep-new(1)`** man page for the complete list) that you will likely want to use when packaging a third-party project include:

`no-version`
: Omit the auto-generated version header. Usually upstream will provide its own equivalent of this functionality.

Note that even if upstream doesn't provide any version information, it's not a good idea to try to rectify this by providing your own version header since upstream may add it in a future version and you may end up with a conflict. Instead, work with the project authors to rectify this upstream.

no-symexport
auto-symexport

The `no-symexport` sub-option suppresses the generation of the DLL symbol exporting header. This is an appropriate option if upstream provides its own symbol exporting arrangements.

The `auto-symexport` sub-option enables automatic DLL symbol exporting support (see Automatic DLL Symbol Exporting for background). This is an appropriate option if upstream relies on similar support in the existing build system. It is also recommended that you give this functionality a try even if upstream does not support building shared libraries on Windows.

binless

Create a header-only library. See Don't make library header-only if it can be compiled and How do I make a header-only C/C++ library?

buildfile-in-prefix

Place header/source `buildfiles` into the header/source prefix directory instead of source subdirectory. To illustrate the difference, compare these two auto-generated layouts paying attention to the location of `buildfiles`:

```
$ bdep new ... --type lib,split,subdir=foo libfoo
$ tree libfoo/
libfoo/
|-- include/
|Â Â  ·-- foo/
|Â Â          |-- buildfile
|Â Â          ·-- foo.hpp
·-- src/
    ·-- foo/
        |-- buildfile
        ·-- foo.cpp

$ bdep new ... --type lib,split,subdir=foo,buildfile-in-prefix libfoo
$ tree libfoo/
libfoo/
|-- include/
|Â Â  |-- foo/
|Â Â  |Â Â  ·-- foo.hpp
|Â Â  ·-- buildfile
·-- src/
    |-- foo/
    |Â Â  ·-- foo.cpp
    ·-- buildfile
```

Note that this sub-option only makes sense if we have the header and/or source prefixes (`include/` and `src/` in our case) as well as the source subdirectory (`foo/` in our case).

> Why would we want to do this? The main reason is to be able to symlink the entire upstream directories rather than individual files. In the first listing, the generated `buildfiles` are inside the `foo/` subdirectories which mean we cannot just symlink `foo/` from upstream.
>
> With a large number of files to symlink, this can be such a strong motivation that it may make sense to invent a source subdirectory in the source prefix even if upstream doesn't have one. See Don't build your main targets in the root `buildfile` for details on this technique.
>
> Another reason we may want to move `buildfiles` to prefix is to be able to handle upstream projects that have multiple source subdirectories. While this situation is not very common in the header prefix, it can be encountered in the source prefix of more complex projects, where upstream wishes to organize the source files into components.

If upstream uses a mixture of C and C++, then it's recommended to set this up using the `--lang` sub-option of `bdep-new`. For example:

```
$ bdep new --lang c++,c ...
```

Continuing with our `libfoo` example, assuming upstream provides its own symbol exporting, the final `bdep-new` command line would be:

```
$ bdep new --package \
  --lang c++,cpp     \
  --type lib,split,subdir=foo,no-subdir-source,no-version,no-symexport \
  libfoo
```

When packaging an executable, things are usually quite a bit simpler: there is no version header, symbol exporting, and the layout is normally combined (since there are no public headers). Typically the only potentially tricky decision you will need to make is whether to use *prefix* or *source subdirectory*. Most likely it will be *prefix* since most executable projects will use the `""` style inclusion for own headers. For example:

```
$ bdep new --package \
  --lang c++         \
  --type exe,no-subdir,prefix=foo,export-stub \
  foo
```

The `export-stub` sub-option causes the generation of `build/export.build`, an export stub that facilitates the importation of targets from our package (see Target Importation for details). The generation of this file for a library is the default since it will normally be used by other projects and thus imported. An executable, however, will only need an export stub if it can plausibly be used during the build (see Build-Time Dependencies and Linked Configurations for background). Source code generators are an obvious example of such executables. A less obvious example would be compression utilities such as `gzip` or `zstd`. If you are unsure, it's best to provide an export stub.

2.2.4 Review and test auto-generated `buildfile` templates

Let's get a more complete view of what got generated by the final `bdep-new` command line from the previous section:

```
$ tree libfoo/
libfoo/
|-- build/
|Â Â  `-- ...
|-- include/
|Â Â  `-- foo/
|Â Â      |-- buildfile
|Â Â      `-- foo.hpp
|-- src/
|Â Â  |-- buildfile
|Â Â  `-- foo.cpp
|-- tests/
|Â Â  |-- build/
|Â Â  |Â Â  `-- ...
|Â Â  |-- basics/
|Â Â  |Â Â  |-- buildfile
|Â Â  |Â Â  `-- driver.cpp
|Â Â  `-- buildfile
|-- buildfile
|-- manifest
`-- README.md
```

Once the overall layout looks right, the next step is to take a closer look at the generated `buildfiles` to make sure that overall they match the upstream build. Of particular interest are the header and source directory `buildfiles` (`libfoo/include/foo/buildfile` and `libfoo/src/buildfile` in the above listing) which define how the library is built and installed.

Here we are focusing on the macro-level differences that are easier to change by tweaking the `bdep-new` command line rather than manually. For example, if we look at the generated source directory `buildfile` and realize it builds a *binful* library (that is, a library that includes source files and therefore produces library binaries) while the upstream library is header-only, it is much easier to fix this by re-running `bdep-new` with the `binless` sub-option than by changing the `buildfiles` manually.

Don't be tempted to start making manual changes at this stage even if you cannot see anything else that can be fixed with a `bdep-new` re-run. This is still a dry-run and we will recreate the package one more time in the following section before starting manual adjustments.

Besides examining the generated `buildfiles`, it's also a good idea to build, test, and install the generated package to make sure everything ends up where you expected and matches upstream where necessary. In particular, make sure public headers are installed into the same location as upstream (unless you have decided to deviate, of course) or at least it's clear how to tweak the generated `buildfiles` to achieve this.

The bdep-new-generated library is a simple "Hello, World!" example that can nevertheless be built, tested, and installed. The idea here is to verify it matches upstream using the generated source files before replacing them with the upstream source file symlinks.

If you are using Windows, then you will need to temporarily replace the no-symexport sub-option with auto-symexport in order to make the generated library buildable. But do not forget to drop this sub-option in the next step.

Note that at this stage it's easiest to build, test, and install in the source directory, skipping the bdep initialization of the package (which we would have to de-initialize before we can re-run bdep-new). Continue with the above example, the recommended sequence of commands would be:

```
$ cd libfoo/
$ b update
$ b test
$ rm -rf /tmp/install
$ b install config.install.root=/tmp/install
$ b clean
```

One relatively common case where the installation location may not match upstream are libraries that include their headers without the subdirectory prefix (for example, <foo_util.h> instead of <foo/util.h>). In such cases, in the bdep-new command, you may want to use *prefix* rather than *source subdirectory* (with the latter being the default). For example:

```
$ bdep new --lib,no-subdir,prefix=foo ...
```

See SOURCE LAYOUT for details.

Let's also briefly discuss other subdirectories and files found in the bdep-new-generated libfoo package.

The build/ subdirectory is the standard build2 place for project-wide build system information (see Project Structure for details). We will look closer at its contents in the following sections.

In the root directory of our package we find the root buildfile and package manifest. We will be tweaking both in the following steps. There is also README.md which we will replace with the upstream symlink.

The tests/ subdirectory is the standard build2 tests subproject (see Testing for background and details). While you can suppress its generation with the no-tests bdep-new sub-option, we recommend that you keep it and use it as a starting point for porting upstream tests or, if upstream doesn't provide any, for a basic "smoke test".

You can easily add/remove/rename this tests/ subproject. The only place where it is mentioned explicitly and where you will need to make changes is the root buildfile. In particular, if upstream provides examples that you wish to port, it is recommended that you use a copy of the generated tests/ subproject as a starting point (not forgetting to add the corresponding entry in the root buildfile).

2.2.5 Create final package

If you are satisfied with the bdep-new command line and there are no more automatic adjustments you can squeeze out of it, then it's time to re-run bdep-new one last time to create the final package.

While redoing this step later will require more effort, especially if you've made manual modifications to buildfile and manifest, nothing is set in stone and it can be done again by simply removing the package directory, removing (or editing, if you have multiple packages and only want to redo some of them) packages.manifest, and starting over.

This time, however, we will do things a bit differently in order to take advantage of some additional automation offered by bdep-new.

Firstly, we will use the special third-party sub-option which is meant for converting third-party projects to build2. Specifically, this sub-option automatically enables no-version and no-symexport (unless auto-symexport is specified). It also adds a number of values to manifest that makes sense to specify in a package of a third-party project. Finally, it generates the PACKAGE-README.md template which describes how to use the package from a build2-based project (see the package-description manifest value for background).

Secondly, if the package directory already exists and contains certain files, bdep-new can take this into account when generating the root buildfile and package manifest. In particular, it will try to guess the license from the LICENSE file and extract the summary from README.md and use this information in manifest as well as generated PACKAGE-README.md.

If the file names or formats used by upstream don't match those recognized by bdep-new, then for now simply omit the corresponding files from the package directory and add them later manually. Similarly, if an attempt to extract the information is unsuccessful, we will have a chance to adjust it in manifest later.

Specifically, for README, bdep-new recognizes README.md, README.txt and README but will only attempt to extract the summary from README.md.

For license files, bdep-new recognizes LICENSE, LICENSE.txt LICENSE.md, COPYING, and UNLICENSE.

For changes-related files, bdep-new recognizes NEWS, CHANGES, and CHANGELOG in various cases as well as with the .md, .txt extensions.

Continuing with our libfoo example and assuming upstream provides the README.md, LICENSE, and NEWS files, we first manually create the package directory, then add the symlinks, and finally run bdep-new (notice that we have replaced no-version and no-symexport with third-party and omitted the package name from the bdep-new command line since we are running from inside the package directory):

```
$ cd foo/ # Change to the package repository root.

$ rm -r libfoo/ packages.manifest
$ mkdir libfoo/

$ cd libfoo/ # Change to the package root.
$ ln -s ../upstream/README.md ./
$ ln -s ../upstream/LICENSE   ./
$ ln -s ../upstream/NEWS      ./

$ bdep new --package \
  --lang c++,cpp     \
  --type lib,split,subdir=foo,no-subdir-source,third-party
```

The final contents of our package will look like this (-> denotes a symlink):

```
$ cd ../
$ tree libfoo/
libfoo/
|-- build/
|Â Â  `-- ...
|-- include/
|Â Â  `-- foo/
|Â Â      |-- buildfile
|Â Â      `-- foo.hpp
|-- src/
|Â Â  |-- buildfile
|Â Â  `-- foo.cpp
|-- tests/
|Â Â  |-- build/
|Â Â  |Â Â  `-- ...
|Â Â  |-- basics/
|Â Â  |Â Â  |-- buildfile
|Â Â  |Â Â  `-- driver.cpp
|Â Â  `-- buildfile
|-- buildfile
|-- manifest
|-- NEWS       -> ../upstream/NEWS
|-- LICENSE    -> ../upstream/LICENSE
|-- README.md  -> ../upstream/README.md
`-- PACKAGE-README.md
```

If auto-detection of `README`, `LICENSE`, and `NEWS` succeeds, then you should see the `summary` and `license` values automatically populated in `manifest` and the symlinked files listed in the root `buildfile`.

2.2.6 Adjust package version

While adjusting the `bdep-new`-generated code is the subject of the following sections, one tweak that we want to make right away is to change the package version in the `manifest` file.

In this guide we will assume the upstream package uses semver (semantic version) or semver-like (that is, has three version components) and will rely on the *continuous versioning* feature of `build2` to make sure that each commit in our package repository has a distinct version (see Versioning and Release Management for background).

If upstream does not use semver, then see How do I handle projects that don't use semantic versioning? and How do I handle projects that don't use versions at all? for available options. If you decide to use the non-semver upstream version as is, then you will have to forgo *continuous versioning* as well as the use of **`bdep-release(1)`** for release management. The rest of the guide, however, will still apply. In particular, you will still be able to use **`bdep-ci(1)`** and **`bdep-publish(1)`** with a bit of extra effort.

The overall plan to implement continuous versioning is to start with a pre-release snapshot of the upstream version, keep it like that while we are adjusting the `bdep-new`-generated package and committing our changes (at which point we get distinct snapshot versions), and finally, when the package is ready to publish, change to the final upstream version with the help of **`bdep-release(1)`**. Specifically, if the upstream version is `*X.Y.Z*`, then we start with the `*X.Y.Z*-a.0.z` pre-release snapshot.

In continuous versioning `*X.Y.Z*-a.0.z` means a snapshot after the (non-existent) `0`th alpha pre-release of the `*X.Y.Z*` version. See Versioning and Release Management for a more detailed explanation and examples.

Let's see how this works for our `libfoo` example. Say, the upstream version that we are packaging is `2.1.0`. This means we start with `2.1.0-a.0.z`.

Naturally, the upstream version that we are using should correspond to the commit of the `upstream` submodule we have added in the Add upstream repository as `git` submodule step.

Next we edit the `manifest` file in the `libfoo` package and change the `version` value to read:

```
version: 2.1.0-a.0.z
```

Let's also commit this initial state of the package for easier rollbacks:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Initialize package"
```

2.3 Fill package with source code and add dependencies

With the package skeleton ready, the next steps are to fill it with upstream source code, add dependencies, and make any necessary manual adjustments to the generated `buildfiles`, `manifest`, etc. If we do this all at once, however, it can be hard to pin-point the cause of build failures. For example, if we convert both the library and its tests right away and something doesn't work, it can be hard to determine whether the mistake is in the library or in the tests. As a result, we are going to split this work into a sequence or smaller steps that incrementally replace the `bdep-new`-generated code with upstream while allowing us to test each change individually. We will also commit the changes on each step for easy rollbacks. Specifically, the overall plan is as follows:

1. Initialize (`bdep-init`) the package in one or more build configurations.
2. Add dependencies, if any.
3. Fill the library with upstream source code.
4. Adjust project-wide and source subdirectory `buildfiles`.
5. Make a smoke test for the library.
6. Replace the smoke test with upstream tests.
7. Tweak root `buildfile` and `manifest`.

The first three steps are the subject of this section with the following sections covering the rest of the plan.

As you become more experienced with packaging third-party projects for `build2`, it may make sense to start combining or omitting some steps, especially for simpler libraries. For example, if you see that a library comes with a simple test that shouldn't cause any complications, then you could omit the smoke test.

2.3.1 Initialize package in build configurations

Before we start making any changes to the `bdep-new`-generated files, let's initialize the package in at least one build configuration so that we are able to build and test our changes (see Getting Started Guide for background on `bdep`-based development workflow). Continuing with our `libfoo` example from the earlier steps:

```
$ cd foo/ # Change to the package repository root.
$ bdep init -C ../foo-gcc @gcc cc config.cxx=g++
```

If you are initializing subsequent packages in the already created configuration, then the command line will be just:

```
$ bdep init @gcc
```

Let's build and test the `bdep-new`-generated package to make sure everything is in order:

```
$ bdep update
$ bdep test
$ bdep clean
```

You can create additional configurations, for example, if you have access to several compilers. For instance, to create a build configuration for Clang:

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

If you would like to perform a certain operation on all the build configurations, pass the `-a|--all` flag to `bdep`:

```
$ bdep update -a
$ bdep test -a
$ bdep clean -a
```

Let's also verify that the resulting package repository is clean (doesn't have any uncommitted or untracked files):

```
$ git status
```

2.3.2 Add dependencies

If the upstream project has any dependencies, now is a good time to specify them so that when we attempt to build the upstream source code, they are already present.

Identifying whether the upstream project has dependencies is not always easy. The natural first places to check are the documentation and the existing build system. Sometimes projects also bundle their dependencies with the project source code (also called vendoring). So it makes sense to look around the upstream repository for anything that looks like bundled dependencies. Normally we would need to "unbundle" such dependencies when converting to `build2` by instead specifying a dependency on an external package (see Don't bundle dependencies for background).

While there are several reasons we insist on unbundling of dependencies, the main one is that bundling can cause multiple, potentially conflicting copies of the same dependency to exist in the build. This can cause subtle build failures that are hard to understand and track down.

One particularly common case to check for is bundling of the testing framework, such as `catch2`, by C++ projects. If you have identified that the upstream tests depend on a testing framework (whether bundled or not), see How do I handle tests that have extra dependencies? for the recommended way to deal with that.

One special type of dependency which is easy to overlook is between packages in the same package repository. For example, if we were packaging both `libfoo` as well as the `foo` executable that depends on it, then the `foo` package has a dependency on `libfoo` and it must be specified. In this case we don't need to add anything to `repositories.manifest` and in the `depends` entry (see below) in `foo`'s `manifest` we will normally use the special `== $` version constraint, meaning `libfoo` should have the same version as `foo` (see the `depends` package `manifest` value for details). For example:

```
depends: libfoo == $
```

If you have concluded that the upstream project doesn't have any dependencies, then you can remove `repositories.manifest` from the package repository root (unless you have already done so), commit this change, and skip the rest of this section.

And if you are still reading, then we assume you have a list of dependencies you need to add, preferably with their minimum required versions. If you could not identify the minimum required version for a dependency, then you can fallback to the latest available version, as will be described in a moment.

With the list of dependencies in hand, the next step is to determine whether they are already available as `build2` packages. For that, head over to cppget.org and search for each dependency.

If you are unable to find a package for a dependency, then it means it hasn't been packaged for `build2` yet. Check the places mentioned in the Check if package repository already exists step to see if perhaps someone is already working on the package. If not and the dependency is not optional, then the only way forward is to first package the dependency.

If you do find a package for a dependency, then note the section of the repository (`stable`, `testing`, etc; see Package Repositories for background) from which the minimum required version of the package is available. If you were unable to identify the minimum required version, then note the latest version available from the `stable` section.

Given the list of repository sections, edit the `repositories.manifest` file in the package repository root and uncomment the entry for `cppget.org`:

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

Next, replace `stable` at the end of the `location` value with the least stable section from your list. For example, if your list contains `stable`, `testing`, and `beta`, then you need `beta` (the sections form a hierarchy and so `beta` includes `testing` which in turn includes `stable`).

If you wish, you can also uncomment the `trust` value and replace `...` with the repository fingerprint. This way you won't be prompted to confirm the repository authenticity on the first fetch. See Adding and Removing Dependencies for details.

Once this is done, edit `manifest` in package root and add the `depends` value for each dependency. See Adding and Removing Dependencies for background. In particular, here you will use the minimum required version (or the latest available) to form a version constraint. Which constraint operator to use will depend on the dependency's versioning policies. If the dependency uses semver, then a `^`-based constraint is a sensible default.

As an example, let's say our `libfoo` depends on `libz`, `libasio`, and `libsqlite3`. To specify these dependencies we would add the following entries to its `manifest`:

```
depends: libz ^1.2.0
depends: libasio ^1.28.0
depends: libsqlite3 ^3.39.4
```

With all the dependencies specified, let's now synchronize the state of the build configurations with our changes by running **bdep-sync(1)** from the package repository root:

```
$ bdep sync -a
```

This command should first fetch the metadata for the repository we specified in `repositories.manifest` and then fetch, unpack and configure each dependency that we specified in `manifest`.

If you have any build-time dependencies (see Build-Time Dependencies and Linked Configurations for background), then you will get a warning about the corresponding `config.import.*` variable being unused and therefore dropped. This is because we haven't yet added the corresponding `import` directives to our `buildfiles`. For now you can ignore this warning, which we will fix later, when we adjust the generated `buildfiles`.

We can examine the resulting state, including the version of each dependency, with **bdep-status(1)**:

```
$ bdep status -ai
```

The last step for this section is to commit our changes:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Add dependencies"
```

2.3.3 Fill with upstream source code

Now we are ready to begin replacing the `bdep-new`-generated files with upstream source code symlinks. We start with the library's header and source files. Continuing with our `libfoo` example, this is what we currently have (notice that `LICENSE`, `README.md`, and `NEWS` are already symlinks to upstream):

```
$ cd foo/ # Change to the package repository root.

$ tree libfoo/
libfoo/
|-- build/
|Â Â  `-- ...
|-- include/
|Â Â  `-- foo/
|Â Â      |-- buildfile
|Â Â      `-- foo.hpp
|-- src/
|Â Â  |-- buildfile
|Â Â  `-- foo.cpp
|-- tests/
|Â Â  `-- ...
|-- buildfile
|-- manifest
|-- NEWS       -> ../upstream/NEWS
|-- LICENSE    -> ../upstream/LICENSE
|-- README.md  -> ../upstream/README.md
`-- PACKAGE-README.md
```

Now we replace generated `include/foo/foo.hpp` with the library's real headers and `src/foo.cpp` with its real source files:

```
$ cd libfoo/ # Change to the package root.

$ cd include/foo/
$ rm foo.hpp
$ ln -s ../../../upstream/include/foo/*.hpp ./
$ cd -

$ cd src/
$ rm foo.cpp
$ ln -s ../../upstream/src/*.hpp ./
$ ln -s ../../upstream/src/*.cpp ./
$ cd -

$ tree libfoo/
libfoo/
|-- build/
|Â Â  `-- ...
|-- include/
|Â Â  `-- foo/
|Â Â      |-- buildfile
|Â Â      |-- core.hpp -> ../../../upstream/include/foo/core.hpp
|Â Â      `-- util.hpp -> ../../../upstream/include/foo/util.hpp
|-- src/
|Â Â  |-- buildfile
|Â Â  |-- impl.hpp -> ../../upstream/src/impl.hpp
|Â Â  |-- core.cpp -> ../../upstream/src/core.cpp
|Â Â  `-- util.cpp -> ../../upstream/src/util.cpp
|-- tests/
|Â Â  `-- ...
`-- ...
```

Note that the wildcards used above may not be enough in all situations and it's a good idea to manually examine the relevant upstream directories and make sure nothing is missing. Specifically, look out for:

- Header/sources with other extensions, for example, C, Objective-C, etc.
- Other files that may be needed, for example, `.def`, `config.h.in`, etc.
- Subdirectories that contain more header/source files.

If upstream contains subdirectories with additional header/source files, then you can symlink entire subdirectories instead of doing it file by file. For example, let's say `libfoo`'s upstream source directory contains the `impl/` subdirectory with additional source files:

```
$ cd src/
$ ln -s ../../upstream/impl ./
$ cd -

$ tree libfoo/
libfoo/
```

```
|-- build/
|Â Â  `-- ...
|-- include/
|Â Â  `-- ...
|-- src/
|Â Â  |-- impl/ -> ../../upstream/src/impl/
|Â Â  |Â Â  |-- bar.cpp
|Â Â  |Â Â  `-- baz.cpp
|Â Â  |-- buildfile
|Â Â  |-- impl.hpp -> ../../upstream/src/impl.hpp
|Â Â  |-- core.cpp -> ../../upstream/src/core.cpp
|Â Â  `-- util.cpp -> ../../upstream/src/util.cpp
|-- tests/
|Â Â  `-- ...
`-- ...
```

Wouldn't it be nice if we could symlink the entire top-level subdirectories (`include/foo/` and `src/` in our case) instead of symlinking individual files? As discussed in Craft `bdep new` command line to create package, we can, but we will need to change the package layout. Specifically, we will need to move the `buildfiles` out of the source subdirectories with the help of the `buildfile-in-prefix` sub-option of `bdep-new`. In the above case, we will also need to invent a source subdirectory in `src/`. Whether this is a worthwhile change largely depends on how many files you have to symlink individually. If it's just a handful, then it's probably not worth the complication, especially if you have to invent source subdirectories. On the other hand, if you are looking at symlinking hundreds of files, changing the layout makes perfect sense.

One minor drawback of symlinking entire directories is that you cannot easily patch individual upstream files (see How do I patch upstream source code?).

You will also need to explicitly list such directories as symlinks in `.gitattributes` if you want your package to be usable from the `git` repository directly on Windows. See Symlinks and Windows for details.

We won't be able to test this change yet because to make things build we will most likely also need to tweak the generated `buildfiles`, which is the subject of the next section. However, it still makes sense to commit our changes to make rollbacks easier:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Add upstream source symlinks"
```

2.4 Adjust project-wide and source `buildfiles`

With source code and dependencies added, the next step is to adjust the regenerated `buildfiles` that build the library. This involves two places: the project-wide build system files in `build/` and the source subdirectory `buildfiles` (in `include/` and `src/` for our `libfoo` example).

2.4.1 Adjust project-wide build system files in `build/`

We start with reviewing and adjusting the files in the `build/` subdirectory of our package, where you will find three files:

```
$ cd foo/ # Change to the package repository root.
$ tree libfoo/
libfoo/
|-- build/
|Â Â  |-- bootstrap.build
|Â Â  |-- root.build
|Â Â  `-- export.build
`-- ...
```

To recap, the first two contain the project-wide build system setup (see Project Structure for details) while the last is an export stub that facilitates the importation of targets from our package (see Target Importation for details).

Normally you don’t need to change anything in `bootstrap.build` – all it does is specify the build system project name and load a standard set of core build system modules. Likewise, `export.build` is ok as generated unless you need to do something special, like exporting targets from different subdirectories of your package.

While `root.build` is also often good as is, situations where you may need to tweak it are not uncommon and include:

- Loading an additional build system module.

  For example, if your package makes use of Objective-C/C++ (see Objective-C Compilation and Objective-C++ Compilation) or Assembler (see Assembler with C Preprocessor Compilation), then `root.build` would be the natural place to load the corresponding modules.

  If your package uses a mixture of C and C++, then it’s recommended to set this up using the `--lang` sub-option of `bdep-new` rather than manually. For example:

  ```
  $ bdep new --lang c++,c ...
  ```

- Specifying package configuration variables.

  If upstream provides the ability to configure their code, for example to enable optional features, then you may want to translate this to `build2` configuration variables, which are specified in `root.build` (see Project Configuration for background and details).

  Note that you don’t need to add all the configuration variables right away. Instead, you could first handle the "core" functionality which doesn’t require any configuration and then add the configuration variables one by one while also making the corresponding changes in `buildfiles`.

> One type of configuration that you should normally not expose when packaging for `build2` is support for both header-only and compiled modes. See Don't make library header-only if it can be compiled for details.

Also, in C++ projects, if you don't have any inline or template files, then you can drop the assignment of the file extension for the `ixx{}` and `txx{}` target types, respectively.

If you have added any configuration variables and would like to use non-default values for some of them in your build, then you will need to reconfigure the package. For example, let's say we have added the `config.libfoo.debug` variable to our `libfoo` package which enables additional debugging facilities in the library. This is how we can reconfigure all our builds to enable this functionality:

```
$ bdep sync -a config.libfoo.debug=true
```

If you have made any changes, commit them (similar to the previous step, we cannot test things just yet):

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Adjust project-wide build system files"
```

2.4.2 Adjust source subdirectory `buildfiles`

The last step we need to perform before we can try to build our library is to adjust its `buildfiles`. These `buildfiles` are found in the source subdirectory or, if we used the `buildfile-in-prefix` `bdep-new` sub-option, in the prefix directory. There will be two `buildfiles` if we use the split layout (`split` sub-option) or a single `buildfile` in the combined layout. The single `buildfile` in the combined layout contains essentially the same definitions as the split `buildfiles` but combined into one and with some minor simplifications that this allows. Here we will assume the split layout and continue with our `libfoo` from the previous sections. To recap, here is the layout we've got with the `buildfiles` of interest found in `include/foo/` and in `src/`:

```
libfoo/
|-- build/
|Â Â  ·-- ...
|-- include/
|Â Â  ·-- foo/
|Â Â      |-- buildfile
|Â Â      |-- core.hpp -> ../../../upstream/include/foo/core.hpp
|Â Â      ·-- util.hpp -> ../../../upstream/include/foo/util.hpp
|-- src/
|Â Â  |-- buildfile
|Â Â  |-- impl.hpp -> ../../upstream/src/impl.hpp
|Â Â  |-- core.cpp -> ../../upstream/src/core.cpp
|Â Â  ·-- util.cpp -> ../../upstream/src/util.cpp
|-- tests/
|Â Â  ·-- ...
·-- ...
```

If instead of a library you are packaging an executable, you can skip directly to Adjust source `buildfile`: executables.

2.4.3 Adjust header `buildfile`

The `buildfile` in `include/foo/` is pretty simple:

The `buildfile` in your package may look slightly different, depending on the exact `bdep-new` sub-options used. However, all the relevant definitions discussed below should still be easily recognizable.

```
pub_hdrs = {hxx ixx txx}{**}

./: $pub_hdrs

# Install into the foo/ subdirectory of, say, /usr/include/
# recreating subdirectories.
#
{hxx ixx txx}{*}:
{
  install         = include/foo/
  install.subdirs = true
}
```

Normally, the only change that you would make to this `buildfile` is to adjust the installation location of headers (see Installing for background). In particular, if our headers were included without the `<foo/...>` prefix but instead contained the library name in their names (for example, `foo-util.hpp`), then the installation setup would instead look like this:

```
# Install directly into say, /usr/include/ recreating subdirectories.
#
{hxx ixx txx}{*}:
{
  install         = include/
  install.subdirs = true
}
```

If the library doesn't have any headers in nested subdirectories (for example, `<foo/util/string.hpp>`), you can drop the `install.subdirs` variable:

```
# Install into the foo/ subdirectory of, say, /usr/include/.
#
{hxx ixx txx}{*}: install = include/foo/
```

In the combined layout, the installation-related definitions are at the end of the combined `buildfile`.

Compared to the split layout where the public and private headers are separated physically, in the combined layout you may need to achieve the same result (that is, avoid installing private headers) at the build system level. If the library provides only a handful of public headers and this set is unlikely to change often, then listing them explicitly is the most straightforward approach. For example (the `@./` qualifier tells `build2` they are in the source directory):

```
# Only install public headers into, say, /usr/include/.
#
h{foo}@./ h{foo_version}@./: install = include/
h{*}: install = false
```

See also How do I handle extra header installation subdirectory?

2.4.4 Adjust source `buildfile`: overview

Next is the `buildfile` in `src/`:

Again, the `buildfile` in your package may look slightly different, depending on the exact `bdep-new` sub-options used. However, all the relevant definitions discussed below should still be easily recognizable.

For a binless (header-only) library, this `buildfile` will contain only a small subset of the definitions shown below. See How do I make a header-only C/C++ library? for additional considerations when packaging header-only libraries.

```
intf_libs = # Interface dependencies.
impl_libs = # Implementation dependencies.
#import xxxx_libs += libhello%lib{hello}

# Public headers.
#
pub = [dir_path] ../include/foo/

include $pub

pub_hdrs = $($pub/ pub_hdrs)

lib{foo}: $pub/{$pub_hdrs}

# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx ixx txx cxx}{**} $impl_libs $intf_libs

# Build options.
#
out_pfx_inc = [dir_path] $out_root/include/
src_pfx_inc = [dir_path] $src_root/include/
out_pfx_src = [dir_path] $out_root/src/
src_pfx_src = [dir_path] $src_root/src/

cxx.poptions =+ "-I$out_pfx_src" "-I$src_pfx_src" \
                "-I$out_pfx_inc" "-I$src_pfx_inc"

#obja{*}: cxx.poptions += -DFOO_STATIC_BUILD
#objs{*}: cxx.poptions += -DFOO_SHARED_BUILD

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_pfx_inc" "-I$src_pfx_inc"
  cxx.export.libs = $intf_libs
}
```

```
#liba{foo}: cxx.export.poptions += -DFOO_STATIC
#libs{foo}: cxx.export.poptions += -DFOO_SHARED

# For pre-releases use the complete version to make sure they cannot
# be used in place of another pre-release or the final version. See
# the version module for details on the version.* variable values.
#
if $version.pre_release
  lib{foo}: bin.lib.version = "-$version.project_id"
else
  lib{foo}: bin.lib.version = "-$version.major.$version.minor"

# Don't install private headers.
#
{hxx ixx txx}{*}: install = false
```

2.4.5 Adjust source `buildfile`: cleanup

As a first step, let's remove all the definitions that we don't need in our library. The two common pieces of functionality that are often not needed are support for auto-generated headers (such as `config.h` generated from `config.h.in`) and dependencies on other libraries.

If you don't have any auto-generated headers, then remove all the assignments and expansions of the `out_pfx_inc` and `out_pfx_src` variables. Here is what the relevant lines in the above `buildfile` should look like after this change:

```
# Build options.
#
src_pfx_inc = [dir_path] $src_root/include/
src_pfx_src = [dir_path] $src_root/src/

cxx.poptions =+ "-I$src_pfx_src" "-I$src_pfx_inc"

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$src_pfx_inc"
}
```

If you do have auto-generated headers, then in the split layout you can remove `out_pfx_inc` if you only have private auto-generated headers and `out_pfx_src` if you only have public ones.

In the combined layout the single `buildfile` does not set the `*_pfx_*` variables. Instead it uses the `src_root` and `out_root` variables directly. For example:

```
# Build options.
#
cxx.poptions =+ "-I$out_root" "-I$src_root"

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_root" "-I$src_root"
}
```

To remove support for auto-generated headers in the combined `buildfile`, simply remove the corresponding `out_root` expansions:

```
# Build options.
#
cxx.poptions =+ "-I$src_root"

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$src_root"
}
```

If you only have private auto-generated headers, then only remove the expansion from `cxx.export.poptions`.

If you don't have any dependencies, then remove all the assignments and expansions of the `intf_libs` and `impl_libs` variables. That is, the following lines in the original `build-file`:

```
intf_libs = # Interface dependencies.
impl_libs = # Implementation dependencies.
#import xxxx_libs += libhello%lib{hello}

# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx ixx txx cxx}{**} $impl_libs $intf_libs

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_pfx_inc" "-I$src_pfx_inc"
  cxx.export.libs = $intf_libs
}
```

Become just these:

```
# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx ixx txx cxx}{**}

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_pfx_inc" "-I$src_pfx_inc"
}
```

2.4.6 Adjust source `buildfile`: dependencies

If you do have dependencies, then let’s handle them now.

Here we will assume dependencies on other libraries, which is the common case. If you have dependencies on executables, for example, source code generators, see Build-Time Dependencies and Linked Configurations on how to handle that. In this case you will also need to reconfigure your package after adding the corresponding `import` directives in order to re-acquire the previously dropped `config.import.*` values. Make sure to also pass any configuration variables you specified in Adjust project-wide build system files in `build/`. For example:

```
$ bdep sync -a --disfigure config.libfoo.debug=true
```

For each library that your package depends on (and which you have added to `manifest` in the Add dependencies step), you need to first determine whether it’s an interface or implementation dependency and then import it either into the `intf_libs` or `impl_libs` variable, respectively.

See Library Exportation and Versioning for background on the interface vs implementation distinction. But as a quick rule of thumb, if the library you are packaging includes a header from the dependency library in one of its public headers, then it’s an interface dependency. Otherwise, it’s an implementation dependency.

Continuing with our `libfoo` example, as we have established in Add dependencies, it depends on `libasio`, `libz`, and `libsqlite3` and let’s say we’ve determined that `libasio` is an interface dependency because it’s included from `include/foo/core.hpp` while the other two are implementation dependencies because they are only included from `src/`. Here is how we would change our `buildfile` to import them:

```
intf_libs = # Interface dependencies.
impl_libs = # Implementation dependencies.
import intf_libs += libasio%lib{asio}
import impl_libs += libz%lib{z}
import impl_libs += libsqlite3%lib{sqlite3}
```

You can tidy this a bit further if you would like:

```
import intf_libs = libasio%lib{asio}
import impl_libs = libz%lib{z}
import impl_libs += libsqlite3%lib{sqlite3}
```

If you don’t have any implementation or interface dependencies, you can remove the assignment and all the expansions of the corresponding `*_libs` variable.

Note also that system libraries like `-lm`, `-ldl` on UNIX or `advapi32.lib`, `ws2_32.lib` on Windows should not be imported. Instead, they should be listed in the `c.libs` or `cxx.libs` variables. See How do I link a system library for details.

2.4.7 Adjust source `buildfile`: public headers

With the unnecessary parts of the `buildfile` cleaned up and dependencies handled, let’s discuss the common changes to the remaining definitions, going from top to bottom. We start with the public headers block:

```
# Public headers.
#
pub = [dir_path] ../include/foo/

include $pub

pub_hdrs = $($pub/ pub_hdrs)

lib{foo}: $pub/{$pub_hdrs}
```

This block gets hold of the list of public headers and makes them prerequisites of the library. Normally you shouldn’t need to make any changes here. If you need to exclude some headers, it should be done in the header `buildfile` in the `include/` directory.

In the combined layout the single `buildfile` does not have such code. Instead, all the headers are covered by the wildcard pattern in the following block.

2.4.8 Adjust source `buildfile`: sources, private headers

The next block deals with sources, private headers, and dependencies, if any:

```
# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx ixx txx cxx}{**} $impl_libs $intf_libs
```

By default it will list all the relevant files as prerequisites of the library, starting from the directory of the `buildfile` and including all the subdirectories, recursively (see Name Patterns for background on wildcard patterns).

If your C++ package doesn’t have any inline or template files, then you can remove the `ixx` and `txx` target types, respectively (which would be parallel to the change made in `root.build`; see Adjust project-wide build system files in `build/`). For example:

```
# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx cxx}{**} $impl_libs $intf_libs
```

Source files other than C/C++ (for example, Assembler, Objective-C/C++) are dealt with in Adjust source `buildfile`: extra requirements below.

The other common change to this block is the exclusion of certain files or making them conditionally included. As an example, let’s say in our `libfoo` the source subdirectory contains a bunch of `*-test.cpp` files which are unit tests and should not be listed as prerequisites of a library. Here is how we can exclude them:

```
# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx cxx}{** -**-test} $impl_libs $intf_libs
```

Let’s also assume our `libfoo` contains `impl-win32.cpp` and `impl-posix.cpp` which provide alternative implementations of the same functionality for Windows and POSIX and therefore should only be included as prerequisites on the respective platforms. Here is how we can handle that:

```
# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx cxx}{** -impl-win32 -impl-posix -**-test}
lib{foo}: cxx{impl-win32}: include = ($cxx.target.class == 'windows')
lib{foo}: cxx{impl-posix}: include = ($cxx.target.class != 'windows')
lib{foo}: $impl_libs $intf_libs
```

There are two nuances in the above example worth highlighting. Firstly, we have to exclude the files from the wildcard pattern before we can conditionally include them. Secondly, we have to always link libraries last. In particular, the following is a shorter but an incorrect version of the above:

```
lib{foo}: {hxx cxx}{** -impl-win32 -impl-posix -**-test} \
          $impl_libs $intf_libs
lib{foo}: cxx{impl-win32}: include = ($cxx.target.class == 'windows')
lib{foo}: cxx{impl-posix}: include = ($cxx.target.class != 'windows')
```

You may also be tempted to use the `if` directive instead of the `include` variable for conditional prerequisites. For example:

```
if ($cxx.target.class == 'windows')
  lib{foo}: cxx{impl-win32}
else
  lib{foo}: cxx{impl-posix}
```

This would also be incorrect. For background and details, see How do I keep the build graph configuration-independent?

2.4.9 Adjust source `buildfile`: build and export options

The next two blocks are the build and export options, which we will discuss together:

```
# Build options.
#
out_pfx_inc = [dir_path] $out_root/include/
src_pfx_inc = [dir_path] $src_root/include/
out_pfx_src = [dir_path] $out_root/src/
src_pfx_src = [dir_path] $src_root/src/

cxx.poptions =+ "-I$out_pfx_src" "-I$src_pfx_src" \
                "-I$out_pfx_inc" "-I$src_pfx_inc"

#obja{*}: cxx.poptions += -DFOO_STATIC_BUILD
#objs{*}: cxx.poptions += -DFOO_SHARED_BUILD

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_pfx_inc" "-I$src_pfx_inc"
  cxx.export.libs = $intf_libs
}

#liba{foo}: cxx.export.poptions += -DFOO_STATIC
#libs{foo}: cxx.export.poptions += -DFOO_SHARED
```

The build options are in effect when the library itself is being built and the exported options are propagated to the library consumers (see Library Exportation and Versioning for background on exported options). For now we will ignore the commented out lines that add `-DFOO_STATIC*` and `-DFOO_SHARED*` macros – they are for symbol exporting and we will discuss this topic separately.

If the library you are packaging only relied on platform-independent APIs, then chances are you won't need to change anything here. On the other hand, if it does anything platform-specific, then you will most likely need to add some options.

As discussed in the Output Directories and Scopes section of the build system introduction, there is a number of variables that are used to specify compilation and linking options, such as `*.poptions` (`cxx.poptions` in the above example), `*.coptions`, etc. The below table shows all of them with their rough `make` equivalents in the third column:

`*.poptions`	preprocess	CPPFLAGS
`*.coptions`	compile	CFLAGS/CXXFLAGS
`*.loptions`	link	LDFLAGS
`*.aoptions`	archive	ARFLAGS
`*.libs`	system libraries	LIBS/LDLIBS

The recommended approach here is to study the upstream build system and copy custom compile/link options to the appropriate `build2` variables. Note, however, that doing it thoughtlessly/faithfully by copying all the options may not always be a good idea. See Which C/C++ compile/link options are OK to specify in a project's buildfile? for the guidelines.

If you are packaging a library that includes a large number of optional features, it may be unclear which of them would make sense to enable by default. The notorious example of this situation is `libsqlite3` which provides hundreds of preprocessor macros to enable or tune various aspects of its functionality.

The recommended approach in cases like this is to study the configuration of such a library in distributions like Debian and Fedora and use the same defaults. In particular, this will allow us to substitute the `build2` package with the system-installed version.

Oftentimes, custom options must only be specified for certain target platforms or when using a certain compiler. While `build2` provides a large amount of information to identify the build configuration as well as more advanced `buildfile` language mechanisms (such as Pattern Matching) to make sense of it, this is a large topic for which we refer you to The `build2` Build System manual. Additionally, github.com/build2-packaging now contains a large number of packages that you can study and search for examples.

While exporting preprocessor macros to communicate configuration is a fairly common technique, it has a number of drawbacks and limitations. Specifically, a large number of such macros will add a lot of noise to the consumer's compilation command lines (especially if multiple libraries indulge in this). Plus, the information conveyed by such macros is limited to simple values and is not easily accessible in consumer `buildfiles`.

To overcome these drawbacks and limitations, `build2` provides a mechanism for conveying metadata with C/C++ libraries (and executables). See, How do I convey additional information (metadata) with executables and C/C++ libraries? for details.

Note that outright replacing the preprocessor macros with metadata can be done if this information is only used by the library consumers. In other words, if the library's public headers rely on the presence of such macros, then we have no choice but to export them, potentially also providing the metadata so that this information is easily accessible from `buildfiles`.

Let's consider a representative example based on our `libfoo` to get a sense of what this normally looks like as well as to highlight a few nuances. We will assume our `libfoo` requires either the `FOO_POSIX` or `FOO_WIN32` macro to be defined during the build in order to identify the target platform. Additionally, extra features can be enabled by defining `FOO_EXTRAS`, which should be done both during the build and for consumption (so this macro must also be exported). Next, this library requires the `-fno-strict-aliasing` compile option for the GCC-class compilers (GCC, Clang, etc). Finally, we need to link `pthread` on POSIX and `ws2_32.lib` on Windows. This is how we would work all this into the above fragment:

```
# Build options.
#
out_pfx_inc = [dir_path] $out_root/include/
src_pfx_inc = [dir_path] $src_root/include/
out_pfx_src = [dir_path] $out_root/src/
src_pfx_src = [dir_path] $src_root/src/

cxx.poptions =+ "-I$out_pfx_src" "-I$src_pfx_src" \
                "-I$out_pfx_inc" "-I$src_pfx_inc"
```

```
cxx.poptions += -DFOO_EXTRAS

if ($cxx.target.class == 'windows')
  cxx.poptions += -DFOO_WIN32
else
  cxx.poptions += -DFOO_POSIX

#obja{*}: cxx.poptions += -DFOO_STATIC_BUILD
#objs{*}: cxx.poptions += -DFOO_SHARED_BUILD

if ($cxx.class == 'gcc')
  cxx.coptions += -fno-strict-aliasing

switch $cxx.target.class, $cxx.target.system
{
  case 'windows', 'mingw32'
    cxx.libs += -lws2_32
  case 'windows'
    cxx.libs += ws2_32.lib
  default
    cxx.libs += -pthread
}

# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_pfx_inc" "-I$src_pfx_inc" -DFOO_EXTRAS
  cxx.export.libs = $intf_libs
}

#liba{foo}: cxx.export.poptions += -DFOO_STATIC
#libs{foo}: cxx.export.poptions += -DFOO_SHARED
```

There are a few nuances in the above code worth keeping in mind. Firstly, notice that we append (rather than assign) to all the non-export variables (`*.poptions`, `*.coptions`, `*.libs`). This is because they may already contain some values specified by the user with their `config.*.*` counterparts. On the other hand, the `*.export.*` variables are assigned.

Secondly, the order in which we append to the variables is important for the value to accumulate correctly. You want to first append all the scope-level values, then target type/pattern-specific, and finally any target-specific; that is, from more general to more specific (see Buildfile Language for background). To illustrate this point, let's say in our `libfoo`, the `FOO_POSIX` or `FOO_WIN32` macro are only necessary when compiling `util.cpp`. Below would be the correct order of assigning to `cxx.poptions`:

```
cxx.poptions =+ "-I$out_pfx_src" "-I$src_pfx_src" \
                "-I$out_pfx_inc" "-I$src_pfx_inc"

cxx.poptions += -DFOO_EXTRAS

#obja{*}: cxx.poptions += -DFOO_STATIC_BUILD
#objs{*}: cxx.poptions += -DFOO_SHARED_BUILD

if ($cxx.target.class == 'windows')
  {obja objs}{util}: cxx.poptions += -DFOO_WIN32
else
  {obja objs}{util}: cxx.poptions += -DFOO_POSIX
```

Note that target-specific `*.poptions` and `*.coptions` must be specified on the object file targets while `*.loptions` and `*.libs` – on the library or executable targets.

2.4.10 Adjust source `buildfile`: symbol exporting

Let's now turn to a special sub-topic of the build and export options that relates to the shared library symbol exporting. To recap, a shared library on Windows must explicitly specify the symbols (functions and global data) that it wishes to make accessible by its consumers (executables and other shared libraries). This can be achieved in three different ways: The library can explicitly mark in its source code the names whose symbols should be exported. Alternatively, the library can provide a `.def` file to the linker that lists the symbols to be exported. Finally, the library can request the automatic exporting of all symbols, which is the default semantics on non-Windows platforms. Note that the last two approaches only work for exporting functions, not data, unless special extra steps are taken by the library consumers. Let's discuss each of these approaches in the reverse order, that is, starting with the automatic symbol exporting.

The automatic symbol exporting is implemented in `build2` by generating a `.def` file that exports all the relevant symbols. It requires a few additional definitions in our `buildfile` as described in Automatic DLL Symbol Exporting. You can automatically generate the necessary setup with the `auto-symexport` `bdep-new` sub-option.

Using a custom `.def` file to export symbols is fairly straightforward: simply list it as a prerequisite of the library and it will be automatically passed to the linker when necessary. For example:

```
# Private headers and sources as well as dependencies.
#
lib{foo}: {hxx cxx}{**} $impl_libs $intf_libs def{foo}
```

Some third-party projects automatically generate their `.def` file. In this case you can try to re-create the same generation in the `buildfile` using an ad hoc recipe (or the `in` or `autoconf` build system modules). If that doesn't look possible (for example, if the generation logic is complex and is implemented in something like Perl or Python), then you can try your luck with automatic symbol exporting. Failing that, the only remaining option is to use a pre-generated `.def` file in the `build2` build.

The last approach is to explicitly specify in the source code which symbols must be exported by marking the corresponding declarations with `__declspec(dllexport)` during the library build and `__declspec(dllimport)` during the library use. This is commonly achieved with a macro, customarily called `*_EXPORT` or `*_API`, which is defined to one of the above specifiers based on whether static or shared library is being built or is being consumed, which, in turn, is also normally signaled with a few more macros, such as `*_BUILD_DLL` and `*_USE_STATIC`.

Because this approach requires extensive changes to the source code, you will normally only use it in your `build2` build if it is already used in the upstream build.

In `build2` you can explicitly signal any of the four situations (shared/static, built/consumed) by uncommenting and adjusting the following four lines in the build and export options blocks:

```
# Build options.
#

...

#obja{*}: cxx.poptions += -DFOO_STATIC_BUILD
#objs{*}: cxx.poptions += -DFOO_SHARED_BUILD

# Export options.
#

...

#liba{foo}: cxx.export.poptions += -DFOO_STATIC
#libs{foo}: cxx.export.poptions += -DFOO_SHARED
```

As an example, let's assume our `libfoo` defines in one of its headers the `FOO_EXPORT` macro based on the `FOO_BUILD_DLL` (shared library is being build) and `FOO_USE_STATIC` (static library is being used) macros that it expects to be appropriately defined by the build system. This is how we would modify the above fragment to handle this setup:

```
# Build options.
#

...

objs{*}: cxx.poptions += -DFOO_BUILD_DLL

# Export options.
#

...

liba{foo}: cxx.export.poptions += -DFOO_USE_STATIC
```

2.4.11 Adjust source `buildfile`: shared library version

The final few lines in the above `buildfile` deal with shared library binary (ABI) versioning:

```
# For pre-releases use the complete version to make sure they cannot
# be used in place of another pre-release or the final version. See
# the version module for details on the version.* variable values.
#
if $version.pre_release
  lib{foo}: bin.lib.version = "-$version.project_id"
else
  lib{foo}: bin.lib.version = "-$version.major.$version.minor"
```

The `bdep-new`-generated setup arranges for the platform-independent versioning where the package's major and minor version components are embedded into the shared library binary name (and `soname`) under the assumption that only patch versions are ABI-compatible.

The two situations where you would want to change this are when the above assumption does not hold and/or when upstream provides platform-specific shared library versions which you would like to re-create in your `build2` build. See Library Exportation and Versioning for background and details.

2.4.12 Adjust source `buildfile`: executables

If instead of a library you are packaging an executable, then, as mentioned earlier, it will most likely be a combined layout with a single `buildfile`. This `buildfile` will also be much simpler compared to the library's. For example, give the following `bdep-new` command:

```
$ bdep new --package \
  --lang c++         \
  --type exe,no-subdir,prefix=foo,export-stub \
  foo
```

The resulting source `buildfile` will look like this:

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

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

out_pfx = [dir_path] $out_root/foo/
src_pfx = [dir_path] $src_root/foo/

cxx.poptions =+ "-I$out_pfx" "-I$src_pfx"
```

If the executable doesn't have any inline/template/header files, then you can remove the `ixx`/`txx`/`hxx` target types, respectively (which would be parallel to the change made in `root.build`; see Adjust project-wide build system files in `build/`). For example:

```
exe{foo}: {hxx cxx}{**} $libs testscript
```

If the source code includes its own headers with the `""` style inclusion (or doesn't have any headers), then we can also get rid of `out_pfx` and `src_pfx`. For example:

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

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

Unfortunately, it's not uncommon for projects that provide both a library and an executable, for the executable source code to include public and/or private library headers with the relative `""` style inclusion. For example:

```
#include "../../libfoo/include/foo/util.hpp"
#include "../../libfoo/src/impl.hpp"
```

This approach won't work in `build2` since the two packages may end up in different directories or the library could even be installed. There are two techniques that can be used to work around this issue (other than patching the upstream source code).

For public headers we can provide, in the appropriate places within the executable package, "thunk headers" with the same names as public headers that simply include the corresponding public header from the library using the <> style inclusion.

For private headers we can provide, again in the appropriate places within the executable package, our own symlinks for a subset of private headers. Note that this will only work if the use of private headers within the executable does not depend on any symbols that are not exported by the library (failing that, the executable will have to always link to the static variant of the library).

For a real example of both of these techniques, see the `zstd` package repository.

Dealing with dependencies in executables is similar to libraries except that here we don't have the interface/implementation distinction; see the Adjust source `buildfile`: dependencies step. For example:

```
import libs = libfoo%lib{foo}

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

Likewise, dealing with build options in executables is similar to libraries except that here we have no export options; see the Adjust source `buildfile`: build and export options step.

If the executable can plausibly be used in a build, then it's recommended to add `build2` metadata as describe in How do I convey additional information (metadata) with executables and C/C++ libraries? See also Modifying upstream source code with C/C++ preprocessor on how to do it without physically modifying upstream source code. See the `zstd` package repository for a real example of doing this.

We will discuss the `testscript` prerequisite in the Make smoke test: executables step below.

2.4.13 Adjust source `buildfile`: extra requirements

The changes discussed so far should be sufficient to handle a typical library or executable that is written in C and/or C++ and is able to handle platform differences with the preprocessor and compile/link options. However, sooner or later you will run into a more complex library that may use additional languages, require more elaborate platform detection, or use additional functionality, such as support for source code generators. The below list provides pointers to resources that cover the more commonly encountered additional requirements.

- The `in` build system module

  Use to process `config.h.in` (or other `.in` files) that don't require Autoconf-style platform probing (`HAVE_*` options).

- The `autoconf` build system module

  Use to process `config.h.in` (or their CMake/Meson variants) that require Autoconf-style platform probing (`HAVE_*` options) or CMake/Meson-specific substitution syntax (`#cmakedefine`, etc).

- Objective-C Compilation and Objective-C++ Compilation

  Use to compile Objective-C (`.m`) or Objective-C++ (`.mm`) source files.

- Assembler with C Preprocessor Compilation

  Use to compile Assembler with C Preprocessor (`.S`) source files.

- Implementing Unit Testing

  Use if upstream has tests (normally unit tests) in the source subdirectory.

- Build-Time Dependencies and Linked Configurations

  Use if upstream relies on source code generators, such as `lex` and `yacc`.

- The `build2` HOWTO

  See the `build2` HOWTO article collection for more unusual requirements.

2.4.14 Test library build

At this point our library should be ready to build, at least in theory. While we cannot build and test the entire package before adjusting the generated `tests/` subproject (the subject of the next step), we can try to build just the library and, if it has any unit tests in the source subdirectory, even run some tests.

If the library is header only, there won't be anything to build unless there are unit tests. Still, you may want to continue with this exercise to detect any syntactic mistakes in the `buildfiles`, etc.

To build only a specific subdirectory of our package, we use the build system directly (continuing with our `libfoo` example):

```
$ cd libfoo/src/ # Change to the source subdirectory.
$ b update
```

If there are any issues, try to fix them and then build again. Once the library builds and if it has unit tests, you can try to run them:

```
$ b test
```

It also makes sense to test the installation and see if anything is off (such as private headers being installed):

```
$ rm -rf /tmp/install
$ b install config.install.root=/tmp/install
```

Once the library builds, it makes sense to commit our changes for easier rollbacks:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Adjust source subdirectory buildfiles"
```

2.5 Make smoke test

With the library build sorted, we need tests to make sure the result is actually functional. As discussed earlier, it is recommended to start with a simple "smoke test", make sure that works, and then replace it with upstream tests. However, if upstream tests look simple enough, you can skip the smoke test. For example, if upstream has all its tests in a single source file and the way it is built doesn't look too complicated, then you can just use that source file in place of the smoke test.

If upstream has no tests, then the smoke test will have to stay. A library can only be published if it has at least one test.

It is also recommended to have the smoke test if upstream tests are in a separate package. See How do I handle tests that have extra dependencies? for background and details.

If instead of a library you are packaging an executable, you can skip directly to Make smoke test: executables.

To recap, the `bdep-new`-generated `tests/` subdirectory looks like this (continuing with our `libfoo` example):

```
libfoo/
|-- ...
`-- tests/
    |-- build/
    |Â Â  |-- bootstrap.build
    |Â Â  `-- root.build
    |-- basics/
    |Â Â  |-- driver.cpp
    |Â Â  `-- buildfile
    `-- buildfile
```

The `tests/` subdirectory is a build system subproject, meaning that it can be built independently, for example, to test the installed version of the library (see Testing for background). In particular, this means it has the `build/` subdirectory with project-wide build system files, the same as the library. The `basics/` subdirectory contains the generated test, which is what we will be turning into a smoke test. The subproject root `buildfile` rarely needs changing.

2.5.1 Adjust project-wide build system files in `tests/build/`

Review and adjust the generated `bootstrap.build` and `root.build` (there will be no `export.build`) similar to the Adjust project-wide build system files in `build/` step.

Here the only change you would normally make is in `root.build` and which is to drop the assignment of extensions for target types that are not used in tests.

2.5.2 Convert generated test to library smoke test

The `basics/` subdirectory contains the `driver.cpp` source file that implements the test and `buildfile` that builds it. You can rename both the test subdirectory (`basics/`) and the source file `driver.cpp`, for example, if you are going with the upstream tests directly. You can also add more tests by simply copying `basics/`.

The purpose of a smoke test is to make sure the library's public headers can be included (including in the installed case, no pun intended), it can be linked, and its basic functionality works.

To achieve this, we modify `driver.cpp` to include the library's main headers and call a few functions. For example, if the library has the initialize/deinitialize type of functions, those are good candidates to call. If the library is not header-only, make sure that the smoke test calls at least one non-inline/template function to test symbol exporting.

Make sure that your test includes the library's public headers the same way as would be done by the library consumers.

Continuing with our `libfoo` example, this is what its smoke test might look like:

```
#include <foo/core.hpp>
#include <foo/util.hpp>

#undef NDEBUG
#include <cassert>

int main ()
{
  foo::context* c (foo::init (0 /* flags */));
  assert (c != nullptr);
  foo::deinit (c);
}
```

The C/C++ `assert()` macro is often adequate for simple tests and does not require extra dependencies. But see How do I correctly use C/C++ assert() in tests?

The test `buildfile` is pretty simple:

```
import libs = libfoo%lib{foo}

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

If you have adjusted the library target name (`lib{foo}`) in the source subdirectory `buildfile`, then you will need to make the corresponding change in the `import` directive here. You may also want to tidy it up by removing unused prerequisite types. For example:

```
import libs = libfoo%lib{foo}

exe{driver}: {hxx cxx}{**} $libs
```

2.5.3 Make smoke test: executables

If instead of a library we are packaging an executable, then instead of the `tests/` subproject we get the `testscript` file in the source subdirectory (see Adjust source `buildfile`: executables for a refresher). This file can be used to write one or more tests that exercise our executable (see Testing for background).

How exactly to test any given executable depends on its functionality. For instance, for a compression utility we could write a roundtrip test that first compresses some input, then decompresses it, and finally compares the result to the original. For example (taken from the `zstd` package repository):

```
: roundtrip
:
echo 'test content' | $* -zc | $* -dc >'test content'
```

On the other hand, for an executable that is a source code generator, proper testing would involve a separate tests package that has a build-time dependency on the executable and that exercises the generated code (see How do I handle tests that have extra dependencies? for background and details). See the `thrift` package repository for an example of this setup.

If the executable provides a way to query its version, one test that you should always be able to write, and which can serve as a last resort smoke test, is the version check. For example:

```
: version
:
$* --version >>~"/EOO/"
/.*$(version.major)\.$(version.minor)\.$(version.patch).*/
EOO
```

See also How do I sanitize the execution of my tests?

2.5.4 Test locally

With the smoke test ready, we can finally do some end-to-end testing of our library build. We will start with doing some local testing to catch basic mistakes and then do the full CI to detect any platform/compiler-specific issues.

First let's run the test in the default build configuration by invoking the build system directly (see Getting Started Guide for background on default configurations):

```
$ cd libfoo/tests/ # Change to the tests/ subproject.
$ b test
```

If there are any issues (compile/link errors, test failures), try to address them and re-run the test.

Once the library builds in the default configuration and the result passes the tests, you can do the same for all the build configurations, in case you have initialized your library in several:

```
$ bdep test -a
```

2.5.5 Test locally: installation

Once the development build works, let's also test the installed version of the library. In particular, this makes sure that the public headers are installed in a way that is compatible with how they are included by our test (and would be included by the library consumers). To test this we first install the library into a temporary directory:

```
$ cd libfoo/ # Change to the package root.
$ rm -rf /tmp/install
$ b install config.install.root=/tmp/install
```

Next we build just the `tests/` subproject out of source and arranging for it to find the installed library (see Output Directories and Scopes for background on the out of source build syntax):

```
$ cd libfoo/ # Change to the package root.
$ b test: tests/@/tmp/libfoo-tests-out/ \
  config.cc.loptions=-L/tmp/install/lib \
  config.bin.rpath=/tmp/install/lib
```

The equivalent MSVC command line would be:

```
> b install config.install.root=c:\tmp\install

> set "PATH=c:\tmp\install\bin;%PATH%"
> b test: tests\@c:\tmp\libfoo-tests-out\^
  config.cc.loptions=/LIBPATH:c:\tmp\install\lib
```

It is a good idea to look over the installed files manually and make sure there is nothing unexpected, for example, missing or extraneous files.

Once done testing the installed case, let's clean things up:

```
$ rm -r /tmp/install /tmp/libfoo-tests-out
```

2.5.6 Test locally: distribution

Another special case worth testing is the preparation of the source distribution (see Distributing for background). This, in particular, is how your package will be turned into the source archive for publishing to cppget.org. Here we are primarily looking for missing files. As a bonus, this will also allow us to test the in source build. First we distribute our package to a temporary directory (again using the default configuration and the build system directly):

```
$ cd libfoo/ # Change to the package root.
$ b dist config.dist.root=/tmp/dist config.dist.uncommitted=true
```

The result will be in the `/tmp/dist/libfoo-<version>/` directory which should resemble our `libfoo/` package but without files like `.gitignore`. Next we build and test the distribution in source:

```
$ cd /tmp/dist/libfoo-<version>/
$ b configure config.cxx=g++
$ b update
$ b test
```

If your package has dependencies that you import in your `buildfile`, then the above `configure` operation will most likely fail because such dependencies cannot be found (it may succeed if they are available as system-installed). The error message will suggest specifying the location of each dependency with the `config.import.*` variable. You can fix this by setting each such `config.import.*` to the location of the default build configuration (created in the Initialize package in build configurations step) which should contain all the necessary dependencies. Simply re-run the `configure` operation until you have discovered and specified all the necessary `config.import.*` variables, for example:

```
$ b configure config.cxx=g++ \
  config.import.libz=.../foo-gcc \
  config.import.libasio=.../foo-gcc \
  config.import.libsqlite3=.../foo-gcc
```

It is a good idea to look over the distributed files manually and make sure there is nothing missing or extraneous.

Once done testing the distribution, let’s clean things up:

```
$ rm -r /tmp/dist
```

2.5.7 Commit and test with CI

With local testing complete, let’s commit our changes and submit a remote CI job to test our library on all the major platforms and with all the major compilers:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Add smoke test"
$ git push -u

$ bdep ci
```

The result of the **`bdep-ci(1)`** command is a link where you can see the status of the builds.

Make sure to wait until there are no more unbuilt configurations (that is, the number of entries with the `<unbuilt>` or `building` result is `0`).

If any builds fail, view the logs to determine the cause, try to fix it, commit your fix, and CI again.

It is possible that upstream does not support some platforms or compilers. For example, it’s common for smaller projects not to bother with supporting "secondary" compilers, such as MinGW GCC on Windows or Homebrew GCC on Mac OS.

If upstream expressly does not support some platform or compiler, it’s probably not worth spending time and energy trying to support it in the package. Most likely it will require changes to upstream source code and that is best done upstream rather than in the package (see Don’t try to fix upstream issues in the package for background). In this case you would want to exclude these platforms/compilers from the CI builds using the `builds` package `manifest` value.

The other common cause of a failed build is a newer version of a compiler or platform that breaks upstream. In this case there are three options: Ideally you would want to fix this in upstream and have a new version released. Failing that, you may want to patch the upstream code to fix the issues, especially if this is one of the major platforms and/or primary compilers (see How do I patch upstream source code? for details). Finally, you can just leave the build failing with the expectation that it will be fixed in the next upstream version. Note that in this case you should not exclude the failing build from CI.

2.6 Replace smoke test with upstream tests

With the smoke test working we can now proceed with replacing it with the upstream tests.

2.6.1 Understand how upstream tests work

While there are some commonalities in how C/C++ libraries are typically built, when it comes to tests there is unfortunately little common ground in how they are arranged, built, and executed. As a result, the first step in dealing with upstream tests is to study the existing build system and try to understand how they work.

If upstream tests prove incomprehensible (which is unfortunately not uncommon) and the only options you see are to go with just the smoke test or to give up, then go with just the smoke test. In this case it's a good idea to create an issue in the package repository mentioning that upstream tests are still a TODO.

If instead of a library you are packaging an executable, then whether the below steps will apply depends on the functionality of the executable.

In particular, testing source code generators would normally involve exercising the generated code, in which case the following will largely apply, though in this case the tests would need to be placed into a separate tests package that has a build-time dependency on the executable (see How do I handle tests that have extra dependencies? for background and details). In fact, if a source code generator is accompanied by a runtime library, then the tests will normally exercise them together (though a runtime library might also have its own tests). See the `thrift` package repository for an example of this setup.

To get you started with analyzing the upstream tests, below are some of the questions you would likely need answered before you can proceed with the conversion:

- **Are upstream tests unit tests or integration tests?**

  While the distinction is often fuzzy, for our purposes the key differentiator between unit and integration tests is which API they use: integration tests only use the library's public API while unit tests need access to the implementation details.

  Normally (but not always), unit tests will reside next to the library source code since they need access to more than just the public headers and the library binary (private headers, individual object files, utility libraries, etc). While integration tests are normally (but again not always) placed into a separate subdirectory, usually called `tests` or `test`.

  If the library has unit tests, then refer to Implementing Unit Testing for background on how to handle them in `build2`.

  If the library has integration tests, then use them to replace (or complement) the smoke test.

  If the library has unit tests but no integration tests, then it is recommended to keep the smoke test since that's the only way the library will be tested via its public API.

- **Do upstream tests use an external testing framework?**

  Oftentimes a C++ library will use an external testing framework to implement tests. Popular choices include `catch2`, `gtest`, `doctest`, and `libboost-test`.

  If a library uses such an external testing framework, then it is recommended to factor tests into a separate package in order to avoid making the library package depend on the testing framework (which is only required during testing). See How do I handle tests that have extra dependencies? for details.

  Sometimes you will find that upstream bundles the source code of the testing framework with their tests. This is especially common with `catch2`. If that's the case, it is strongly recommended that you "unbundle" it by making it a proper external dependency. See Don't bundle dependencies for background.

- **Are upstream tests in a single or multiple executables?**

  It's not unusual for libraries to have a single test executable that runs all the test cases. This is especially common if a C++ testing framework is used. In this case it is natural to replace the contents of the smoke test with the upstream source code, potentially renaming the test subdirectory (`basics/`) to better match upstream naming.

  If upstream has multiple test executables, then they could all be in a single test subdirectory (potentially reusing some common bits) or spread over multiple subdirectories. In both cases it's a good idea to follow the upstream structure unless you have good reasons to deviate. In the former case (all executables in the same subdirectory), you can re-purpose the smoke test subdirectory. In the latter case (each executable in a separate subdirectory) you can make copies of the smoke test subdirectory.

- **Do upstream tests use an internal utility library?**

  If there are multiple test executables and they need to share some common functionality, then it's not unusual for upstream to place such functionality into a static library and then link it to each test executable. In `build2` such an internal library is best represented with a utility library (see Implementing Unit Testing for details). See the following section for an example.

- **Are upstream tests well behaved?**

  Unfortunately, it's not uncommon for upstream tests not to behave well, such as to write diagnostics to `stdout` instead of `stderr`, create temporary files without cleaning them up, or assume presence of input files in the current working directory. For details on how to deal with such situations see How do I sanitize the execution of my tests?

2.6.2 Convert smoke test to upstream tests

Once you have a good grasp of how upstream tests work, convert or replace the smoke test with the upstream tests. If upstream has multiple test executables, you may want to deal with one test at a time, making sure that it passes before moving to the next one.

It's normally a good idea to use the smoke test `buildfile` as a starting point for upstream tests. To recap, the smoke test `buildfile` for our `libfoo` example ended up looking like this:

```
import libs = libfoo%lib{foo}

exe{driver}: {hxx cxx}{**} $libs
```

At a minimum you will most likely need to change the name of the executable to match upstream. If you need to build multiple executables in the same directory, then it's probably best to get rid of the name pattern for the source files and specify the prerequisite names explicitly, for example:

```
import libs = libfoo%lib{foo}

./: exe{test1}: cxx{test1} $libs
./: exe{test2}: cxx{test2} $libs
```

If you have a large number of such test executables, then a `for`-loop might be a more scalable option:

```
import libs = libfoo%lib{foo}

for src: cxx{test*}
  ./: exe{$name($src)}: $src $libs
```

If the upstream tests have some common functionality that is used by all the test executables, then it is best placed into a utility library. For example:

```
import libs = libfoo%lib{foo}

./: exe{test1}: cxx{test1} libue{common}
./: exe{test2}: cxx{test2} libue{common}

libue{common}: {hxx cxx}{common} $libs
```

2.6.3 Test locally

With the upstream tests ready, we re-do the same end-to-end testing as we did with the smoke test:

Test locally
Test locally: installation
Test locally: distribution

2.6.4 Commit and test with CI

With local testing complete, we commit our changes and submit a remote CI job. This step is similar to what we did for the smoke test but this time we are using the upstream tests:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Add upstream tests"
$ git push

$ bdep ci
```

2.7 Add upstream examples, benchmarks, if any

If the upstream project provides examples and/or benchmarks and you wish to add them to the `build2` build (which is not strictly necessary for the `build2` package to be usable), then now is a good time to do that.

As was mentioned in Review and test auto-generated `buildfile` templates, the recommended approach is to copy the `tests/` subproject (potentially from the commit history before the smoke test was replaced with the upstream tests) and use that as a starting point for examples and/or benchmarks. Do not forget to add the corresponding entry in the root `buildfile`.

Once that is done, follow the same steps as in Replace smoke test with upstream tests to add upstream examples/benchmarks and test the result.

2.8 Adjust root files (`buildfile`, `manifest`, etc)

The last few files that we need to review and potentially adjust are the root `buildfile`, package `manifest`, and `PACKAGE-README.md`.

2.8.1 Adjust root `buildfile`

The main function of the root `buildfile` is to pull in all the subdirectories that need building plus list targets that are usually found in the root directory of a project, typically `README.md`, `LICENSE`, etc. This is what the generated root `buildfile` looks like for our `libfoo` project assuming we have symlinked `README.md`, `LICENSE`, and `NEWS` from upstream in the Create final package step:

```
./: {*/ -build/}                          \
    doc{README.md PACKAGE-README.md NEWS} \
    legal{LICENSE} manifest

# Don't install tests.
#
tests/: install = false
```

If the upstream project provides any other documentation (detailed change logs, contributing guidelines, etc) or legal files (alternative licenses, list of authors, code of conduct, etc), then you may want to symlink and list them as the `doc{}` and `legal{}` prerequisites, respectively.

If you are packaging an executable and it provides a man page, then it can also be listed in the root `buildfile`. For example, if the man page file is called `foo.1`:

```
./: ... man1{foo}
```

One file you don't need to list is `INSTALL` (or equivalent) which normally contains the installation instructions for the upstream build system. In the `build2` package of a third-party project the `PACKAGE-README.md` file serves this purpose (see Adjust `PACKAGE-README.md` for details).

2.8.2 Adjust root `buildfile`: other subdirectories

If the upstream project has other subdirectories that makes sense to include into the `build2` package, then now is a good time to take care of that. The most common such case will be extra documentation (besides the root `README`), typically in a subdirectory called `doc/`, `docs/`, or `documentation/`.

The standard procedure for handling such subdirectories will be to symlink the relevant files (or the entire subdirectory) and then list the files as prerequisites. For this last step, there are two options: we can list the files directly in the root `buildfile` or we can create a separate `buildfile` in the subdirectory.

If symlinking entire subdirectories, don't forget to also list them in `.gitattributes` if you want your package to be usable from the `git` repository directly on Windows. See Symlinks and Windows for details.

Let's examine each approach using our `libfoo` as an example. We will assume that the upstream project contains the `docs/` subdirectory with additional `*.md` files that document the library's API. It would make sense to include them into the `build2` package.

Listing the subdirectory files directly in the root `buildfile` works best for simple cases, where you have a bunch of static files that don't require any special provisions, such as customizations to their installation locations. In this case we can symlink the entire `docs/` subdirectory:

```
$ cd libfoo/ # Change to the package root.
$ ln -s ../upstream/docs ./
```

The adjustments to the root `buildfile` are pretty straightforward: we exclude the `docs/` subdirectory (since it has no `buildfile`) and list the `*.md` files as prerequisites using the `doc{}` target type (which, in particular, makes sure they are installed into the appropriate location):

```
./: {*/ -build/ -docs/}                    \
    doc{README.md PACKAGE-README.md NEWS} \
    docs/doc{*.md}                        \
    legal{LICENSE} manifest
```

The alternative approach (create a separate `buildfile`) is a good choice if things are more complicated than that. Let's say we need to adjust the installation location of the files in `docs/` because there is another `README.md` inside and that would conflict with the root one when installed into the same location. This time we cannot symlink the top-level `docs/` subdirectory (because we need to place a `buildfile` there). The two options here are to either symlink the individual files or introduce another subdirectory level inside `docs/` (which is the same approach as discussed in Don't build your main targets in the root `buildfile`). Let's illustrate both sub-cases.

Symlinking individual files works best when you don't expect the set of files to change often. For example, if `docs/` contains a man page and its HTML rendering, then it's unlikely this set will change. On the other hand, if `docs/` contains a manual split into an `.md` file per chapter, then there is a good chance this set of files will fluctuate between releases.

Continuing with our `libfoo` example, this is how we symlink the individual `*.md` files in `docs/`:

```
$ cd libfoo/ # Change to the package root.
$ mkdir docs
$ cd docs/
$ ln -s ../../upstream/docs/*.md ./
```

Then write a new `buildfile` in `docs/`:

```
./: doc{*.md}

# Install the documentation in docs/ into the manual/ subdirectory of,
# say, /usr/share/doc/libfoo/ since we cannot install both its and root
# README.md into the same location.
#
doc{*.md}: install = doc/manual/
```

Note that we don't need to make any changes to the root `buildfile` since this subdirectory will automatically get picked up by the `{*/ -build/}` name pattern that we have there.

Let's now look at the alternative arrangement with another subdirectory level inside `docs/`. Here we achieve the same result but in a slightly different way. Specifically, we call the subdirectory `manual/` and install recreating subdirectories (see Installing for background):

```
$ cd libfoo/ # Change to the package root.
$ mkdir -p docs/manual
$ cd docs/manual/
$ ln -s ../../../upstream/docs/*.md ./
```

And the corresponding `buildfile` in `docs/`:

```
./: doc{**.md}

# Install the documentation in docs/ into, say, /usr/share/doc/libfoo/
# recreating subdirectories.
#
doc{*}:
{
  install = doc/
  install.subdirs = true
}
```

Yet another option would be to open a scope for the `docs/` subdirectory directly in the root `buildfile` (see Output Directories and Scopes for background). For example:

```
$ cd libfoo/ # Change to the package root.
$ ln -s ../upstream/docs ./
```

And then add the following to the root `buildfile`:

```
docs/
{
  ./: doc{*.md}

  # Install the documentation in docs/ into the manual/ subdirectory
  # of, say, /usr/share/doc/libfoo/ since we cannot install both its
  # and root README.md into the same location.
  #
  doc{*.md}: install = doc/manual/
}
```

However, this approach should be used sparingly since it can quickly make the root `build-file` hard to comprehend. Note also that it cannot be used for main targets since an export stub requires a `buildfile` to load (see Don't build your main targets in the root `build-file` for details).

2.8.3 Adjust root `buildfile`: commit and test

Once all the adjustments to the root `buildfile` are made, it makes sense to test it locally (this time from the root of the package), commit our changes, and test with CI:

```
$ cd libfoo/ # Change to the package root.
$ b test
$ bdep test -a
```

If you had to add any extra files to the root `buildfile` (or add `buildfiles` in extra subdirectories), then it also makes sense to test the installation (Test locally: installation) and the preparation of the source distribution (Test locally: distribution) to make sure the extra files end up in the right places.

Then commit our changes and CI:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Adjust root buildfile"
$ git push

$ bdep ci
```

2.8.4 Adjust `manifest`

The next file we need to look over is the package's `manifest`. Here is what it will look like, using our `libfoo` as an example:

```
: 1
name: libfoo
version: 2.1.0-a.0.z
language: c++
project: foo
summary: C++ library implementing secure Foo protocol
license: MIT ; MIT License.
description-file: README.md
package-description-file: PACKAGE-README.md
changes-file: NEWS
url: https://example.org/upstream
email: upstream@example.org
package-url: https://github.com/build2-packaging/foo
package-email: packaging@build2.org ; Mailing list.
depends: * build2 >= 0.16.0
depends: * bpkg >= 0.16.0
```

You can find the description of these and other package `manifest` values in Package Manifest (the manifest format is described in Manifest Format).

In the above listing the values that we likely need to adjust are `summary` and `license`, unless correctly auto-detected by `bdep-new` in the Create final package step. See Adjust `manifest`: `summary` and Adjust `manifest`: `license` below for guidelines on changing these values.

It is not uncommon for projects to be licensed under multiple licenses. Note, however, that `bdep-new` will only detect one license and you will need to specify any additional licenses manually.

We will also need to change `url` and `email` with the upstream project's homepage URL and e-mail, respectively. If upstream doesn't have a dedicated website for the project, then use its repository URL on GitHub or equivalent. For e-mail you would normally use a mailing list address. If upstream doesn't have any e-mail contacts, then you can drop this value from the `manifest`. The `package-url` and `package-email` values normally do not need to be changed.

packaging@build2.org is a mailing list for discussions related to the packaging efforts of third-party projects.

Note also that while you may be tempted to adjust the `version` value, resist this temptation since this will be done automatically by **`bdep-release(1)`** later.

You may also want to add the following values in certain cases:

`changes-file`
If you have added any extra news of changelog files to the root `buildfile` (see Adjust root buildfile), then it may also make sense to list them in the `manifest`. For example:

```
changes-file: ChangeLog.txt
```

`topics`
Package topics. For example:

```
topics: network protocol, network security
```

If the upstream project is hosted on GitHub or similar, then you can usually copy the topics from the upstream repository description.

`doc-url`
`src-url`
Documentation and source code URLs. For example:

```
doc-url: https://example.org/foo/doc/
src-url: https://github.com/.../foo
```

2.8.5 Adjust `manifest`: `summary`

For `summary` use a brief description of the functionality provided by the library or executable. Less than 70 characters is a good target to aim for. Don’t capitalize subsequent words unless proper nouns and omit the trailing dot. For example:

```
summary: Vim xxd hexdump utility
```

Omit weasel words such as "modern", "simple", "fast", "small", etc., since they don’t convey anything specific. Omit "header-only" or "single-header" for C/C++ libraries since, at least in the context of `build2`, it does not imply any benefit.

If upstream does not offer a sensible summary, the following template is recommended for libraries:

```
summary: <functionality> C library
summary: <functionality> C++ library
```

For example:

```
summary: Event notification C library
summary: Validating XML parsing and serialization C++ library
```

If the project consists of multiple packages, it may be tempting to name each package in terms of the overall project name, for example:

```
name: libigl-core
summary: libigl core module
```

This doesn't give the user any clue about what functionality is provided unless they find out what `libigl` is about. Better:

```
summary: Geometry processing C++ library, core module
```

If you follow the above pattern, then to produce a summary for external tests or examples packages simply add "tests" or "examples" at the end, for example:

```
summary: Event notification C library tests
summary: Geometry processing C++ library, core module examples
```

2.8.6 Adjust `manifest`: `license`

For `license`, use the SPDX license ID if at all possible. If multiple licenses are involved, use the SPDX License expression. See the `license` manifest value documentation for details, including the list of the SPDX IDs for the commonly used licenses.

2.8.7 Adjust `manifest`: commit and test

Once all the adjustments to the `manifest` are made, it makes sense to test it locally, commit our changes, and test with CI:

```
$ cd libfoo/ # Change to the package root.
$ b test
$ bdep test -a
```

Then commit our changes and CI:

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Adjust manifest"
$ git push

$ bdep ci
```

2.8.8 Adjust `PACKAGE-README.md`

The last package file we need to adjust is `PACKAGE-README.md` which describes how to use the package from a `build2`-based project. The template generated by `bdep-new` establishes the recommended structure and includes a number of placeholders enclosed in `< >`, such as `<UPSTREAM-NAME>` and `<SUMMARY-OF-FUNCTIONALITY>`, that need to be replaced with the package-specific content. While all the placeholders should be self-explanatory, below are a couple of guidelines.

For `<SUMMARY-OF-FUNCTIONALITY>` it's best to copy a paragraph or two from the upstream documentation, usually from `README.md` or the project's web page.

If the `bdep new` command was able to extract the summary from upstream `README`, then the summary in the heading (first line) will contain that information. Otherwise, you would need to adjust it manually, similar to `manifest` above. In this case use the `summary` value form the `manifest`, perhaps slightly shortened.

If the package contains a single importable target, as is typical with libraries, then it makes sense to drop the "Importable targets" section since it won't add anything that hasn't already been said in the "Usage" section.

Similarly, if the package has no configuration variables, then it makes sense to drop the "Configuration variables" section.

For inspiration, see

`PACKAGE-README.md` in `zstd` and `PACKAGE-README.md` in `libevent` (libraries) as well as `PACKAGE-README.md` in `zstd` and `README.md` in `xxd` (executables).

If upstream does not provide a `README` file, then it makes sense to rename `PACKAGE-README.md` to just `README.md` in the `build2` package, as was done in the `xxd` package mentioned above.

Once `PACKAGE-README.md` is ready, commit and push the changes. You may also want to view the result on GitHub to make sure everything is rendered correctly.

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Adjust PACKAGE-README.md"
$ git push
```

2.9 Adjust package repository `README.md`

With all the package files taken care of, the last file we need to adjust is `README.md` in the root of our package repository (it was created in the Initialize package repository with `bdep new` step).

If you need to add additional packages and are doing this one package at a time (for example, first library then executable in the "library and executable" project), then this is the point where you would want to restart from Create package and generate `buildfile` templates for another iteration. Only once all the packages are added does it make sense to continue with updating this `README.md`.

The primary purpose of the package repository `README.md` is to provide setup instructions as well as any other relevant information for the development of the packages as opposed to their consumption. However, it's also a good idea to give a brief summary of what this repository is about and to point users interested in consumption to the `PACKAGE-README.md` files.

The template generated by `bdep new` establishes the recommended structure to achieve these objectives. It includes a number of placeholders enclosed in < >, such as `<UPSTREAM-URL>` and `<SUMMARY-OF-FUNCTIONALITY>`, that need to be replaced with the repository-specific content. While all the placeholders should be self-explanatory, below are a couple of guidelines.

If there is a single package, then `<SUMMARY>` in the heading can be the same as in `PACKAGE-README.md`. If there are multiple packages, then use an overall summary of the upstream project.

For `<SUMMARY-OF-FUNCTIONALITY>` it's best to copy a paragraph or two from the upstream documentation, usually from `README.md` or the project's web page. Again, for a single package, this can be copied directly from `PACKAGE-README.md`.

If there are multiple packages in the repository, then it's recommended to replace a single link to `PACKAGE-README.md` with a list of links (this also shows the available packages). For example:

```
... If you want to use `foo` in your `build2`-based project, then
instead see the accompanying `PACKAGE-README.md` files:

* [`libfoo/PACKAGE-README.md`](libfoo/PACKAGE-README.md)
* [`foo/PACKAGE-README.md`](foo/PACKAGE-README.md)
```

The remainder of the generated `README.md` file are the standard `bdep` initialization instructions. Adjust them if your package repository requires anything special (for example, a host configuration). This is also the place to mention anything unusual, such as that upstream does not use semver (and thus only a subset of `bdep` functionality is usable).

For inspiration, see `README.md` in the `zstd` package repository.

Once the repository `README.md` is ready, commit and push the changes. You may also want to view the result on GitHub to make sure everything is rendered correctly.

```
$ cd foo/ # Change to the package repository root.
$ git add .
$ git status
$ git commit -m "Adjust package repository README.md"
$ git push
```

2.10 Release and publish

Once all the adjustments are in and everything is tested, we can release the final version of the package and then publish it to cppget.org. Both of these steps are automated with the corresponding `bdep` commands. But before performing these steps we need to transfer the package repository to github.com/build2-packaging.

2.10.1 Transfer package repository

If you have been doing your work in a repository in your personal workspace, then now is the time to transfer it to the github.com/build2-packaging organization.

It is important to transfer the repository before publishing the first version of the package since the repository is used as a proxy for package name ownership (see **`bdep-publish(1)`** for details). If you publish the package from your personal workspace and then transfer the repository, the ownership information will have to be adjusted manually, which we would prefer to avoid.

The first step is to become a member of this organization (unless you already are). This will give you permissions to create new repositories, which is required to perform a transfer (you will also have full read/write access to the repository once transferred). To get an invite, get in touch not forgetting to mention your GitHub user name.

If your repository has any prefixes, such as `build2-`, or suffixes such as `-package`, then the next step is to rename it to follow the Use upstream repository name as package repository name guideline. Go to the repository's Settings on GitHub where you should see the Rename button.

Finally, to perform the transfer, go to the repository's Settings, Danger Zone section, where you should see the Transfer button. Select `build2-packaging` as the organization to transfer to, and complete the transfer.

Once transferred, you will be considered the maintainer of this package going forward. If other members of the `build2-packaging` organization wish to participate in the package maintenance, the correct etiquette is to do this via pull requests. However, if you lose interest in maintaining a package or otherwise become unresponsive, we may allow a new maintainer to take over this role.

In extraordinary circumstances the `build2-packaging` administrators may make direct changes to the package, for example, to release a new revision in order to address a critical issue. They will still try to coordinate the changes with the maintainer but may not always be able to wait for a response in time-sensitive cases.

2.10.2 Release final version

As you may recall, our package currently has a pre-release snapshot version of the upstream version (see Adjust package version). Once all the changes are in, we can change to the final upstream version, in a sense signaling that this package version is ready.

If you are working in a branch, then now is also the time to merge it into `master` (or equivalent).

The recommended way to do this is with the **`bdep-release(1)`** command (see Versioning and Release Management for background). Besides replacing the `version` value in the package `manifest` file, it also commits this change, tags it with the `v`*`X`*`.`*`Y`*`.`*`Z`* tag, and can be instructed to push the changes (or show the `git` command to do so). This command also by default "opens" the next development version, which is something that we normally want for our own projects but not when we package a third-party one (since we cannot predict which version upstream will release next). So we disable this functionality. For example:

```
$ cd foo/ # Change to the package repository root.
$ bdep release --no-open --show-push
```

Then review the commit made by `bdep-release` and, if everything looks good, push the changes by copying the command that it printed:

```
$ git diff HEAD~1
$ git push ...
```

If something is wrong and you need to undo this commit, don’t forget to also remove the tag. Note also that once you have pushed your changes, you cannot undo the commit. Instead, you will need to make a revision. See Version management for background and details.

2.10.3 Publish released version

Once the version is released we can publish the package to cppget.org with the **`bdep-publish(1)`** command (see Versioning and Release Management for background):

```
$ cd foo/ # Change to the package repository root.
$ bdep publish
```

The `bdep-publish` command prepares the source distribution of your package, uploads the resulting archive to the package repository, and prints a link to the package submission in the queue. Open this link in the browser and check that there are no surprises in the build results (they should match the earlier CI results) or in the displayed package information (`PACKAGE-README.md`, etc).

While there should normally be no discrepancies in the build results compared to our earlier CI submissions, the way the packages are built on CI and in the package repository are not exactly the same. Specifically, CI builds them from `git` while the package repository – from the submitted package archives. If there are differences, it’s almost always due to issues in the source distribution preparation (see Test locally: distribution).

If everything looks good, then you are done: the package submission will be reviewed and, if there are no problems, moved to cppget.org. If there are problems, then an issue will be created in the package repository with the review feedback. In this case you will need to release and publish a version revision to address any problems. However, in both cases, you should first read through the following Package version management section to understand the recommended "version lifecycle" of a third-party package.

Also, if there is an issue for this package in github.com/build2-packaging/WISHLIST, then you would want to add a comment and close it once the package has been moved to cppget.org.

2.11 Package version management

Once we have pushed the release commit, in order to preserve continuous versioning (see Adjust package version for background), no further changes should be made to the package without also changing its version.

More precisely, you can make and commit changes without changing the version provided they don't affect the package. For example, you may keep a `TODO` file in the root of your repository which is not part of any package. Updating such a file without changing the version is ok since the package remains unchanged.

While in our own projects we can change the versions as we see fit, with third-party projects the versions are dictated by upstream and as a result we are limited to what we can use to fix issues in our packaging work itself. It may be tempting (and perhaps even conceptually correct) to release a patch version for our own fixes, however, we will be in trouble if later upstream releases the same patch version but with a different set of changes (plus the users of our package may wonder where did this version come from). As a result, we should only change the major, minor, or patch components of the package version in response to the corresponding upstream releases. For fixes to the packaging work itself we should instead use version revisions.

Because a revision replaces the existing version, we should try to limit revision changes to bug fixes and preferably only in the package "infrastructure" (`buildfiles`, `manifest`, etc). Fixes to upstream source code should be limited to critical bugs and be preferably backported from upstream. To put it another way, changes in a revision should have an even more limited scope than a patch release.

Based on this, the recommended "version lifecycle" for a third-party package is as follows:

1. After a release (the Release final version step above), for example, version `2.1.0`, the package enters a "revision phase" where we can release revisions (`2.1.0+1`, `2.1.0+2`, etc) to address any issues in the packaging work. See New revision for the detailed procedure.
2. When a new upstream version is released, for example version `2.2.0`, and we wish to upgrade our package to this version, we switch to its pre-release snapshot version (`2.2.0-a.0.z`) the same way as we did in the Adjust package version step initially. See New version for the detailed procedure.
3. Once we are done upgrading to the new upstream version, we release the final version just like in the Release final version step initially. At this point the package enters another revision phase.

Note also that in the above example, once we have switched to `2.2.0-a.0.z`, we cannot go back and release another revision or patch version for `2.1.0` on the current branch. Instead, we will need to create a separate branch for the `2.1.Z` release series and make a revision or patch version there. See New version/revision in old release series for the detailed procedure.

2.11.1 New revision

As discussed in Package version management, we release revisions to fix issues in the package "infrastructure" (`buildfiles`, `manifest`, etc) as well as critical bugs in upstream source code.

Releasing a new revision is also a good opportunity to review and fix any accumulated issues that didn't warrant a revision on their own. See New version: review/fix accumulated issues for background.

In the revision phase of the package version lifecycle (i.e., when the version does not end with `-a.0.z`), every commit must be accompanied by the revision increment to maintain continuous versioning. As a result, each revision release commit necessarily also contains the changes in this revision. Below is a typical workflow for releasing and publishing the revision:

```
$ # make changes
$ # test locally
$ git add .
$ bdep release --revision --show-push
$ # review commit
$ git push ...
$ # test with CI
$ bdep publish
```

Customarily, the revision commit message has the `"Release version X.Y.Z+R"` summary as generated by `bdep-release` followed by the description of changes, organized in a list if there are several. For example:

```
Release version 2.1.0+1

- Don't compile port/strlcpy.c on Linux if GNU libc is 2.38 or newer
  since it now provides the strl*() functions.

- Switch to using -pthread instead of -D_REENTRANT/-lpthread.
```

The fact that all the changes must be in a single commit is another reason to avoid substantial changes in revisions.

Note also that you can make multiple commits while developing and testing the changes for a revision in a separate branch. However, once they are ready for a release, they need to be squashed into a single commit. The **`bdep-release(1)`** command provides the `--amend` and `--squash` options to automate this. For example, here is what a workflow with a separate branch might look like:

```
$ git checkout -b wip-2.1.0+1

$ # make strl*() changes
$ # test locally
$ git commit -a -m "Omit port/strlcpy.c if glibc 2.38 or newer"
$ git push -u
$ # test with CI

$ # make pthread changes
$ # test locally
$ git commit -a -m "Switch to using -pthread"
$ git push
$ # test with CI

$ git checkout master
$ git merge --ff-only wip-2.1.0+1
$ bdep release --revision --show-push --amend --squash 2
$ # review commit
$ # test locally
$ git push ...
$ # test with CI
$ bdep publish
```

2.11.2 New version

As discussed in Package version management, we release new versions strictly in response to the corresponding upstream releases.

The amount or work required to upgrade a package to a new upstream version depends on the extend of changes in the new version.

On one extreme you may have a patch release which fixes a couple of bugs in the upstream source code without any changes to the set of source files, upstream build system, etc. In such cases, upgrading a package is a simple matter of creating a new work branch, pointing the `upstream` `git` submodule to the new release, running tests, and then merging, releasing, and publishing a new package version.

On the other extreme you may have a new major upstream release which is essentially a from-scratch rewrite with new source code layout, different upstream build system, etc. In such cases it may be easier to likewise start from scratch. Specifically, create a new work branch, point the `upstream` `git` submodule to the new release, delete the existing package, and continue from Create package and generate `buildfile` templates.

Most of the time, however, it will be something in between where you may need to tweak a few things here and there, such as adding symlinks to new source files (or removing old ones), tweaking the `buildfiles` to reflect changes in the upstream build system, etc.

The following sections provide a checklist-like sequence of steps that can be used to review upstream changes with links to the relevant earlier sections in case adjustments are required.

2.11.3 New version: create new work branch

When upgrading a package to a new upstream version it's recommended to do this in a new work branch which, upon completion, is merged into `master` (or equivalent). For example, if the new upstream version is `2.2.0`:

```
$ git checkout -b wip-2.2.0
```

If you are not the maintainer of the package and would like to help with preparing the new version, then, when everything is ready, use this branch to create a pull request instead of merging it directly.

2.11.4 New version: open new version

This step corresponds to Adjust package version during the initial packaging. Here we can make use of the `bdep-release` command to automatically open the new version and make the corresponding commit. For example, if the new upstream version is `2.2.0`:

```
$ bdep release --open --no-push --open-base 2.2.0
```

2.11.5 New version: update `upstream` submodule

This step corresponds to Add upstream repository as `git` submodule during the initial packaging. Here we need to update the submodule to point to the upstream commit that corresponds to the new version.

For example, if the upstream release tag we are interested in is called `v2.2.0`, to update the `upstream` submodule to point to this release commit, run the following commands:

```
$ cd upstream/
$ git fetch
$ git checkout v2.2.0
$ cd ../

$ git add .
$ git status
$ git commit -m "Update upstream submodule to 2.2.0"
```

2.11.6 New version: review upstream changes

At this point it's a good idea to get an overview of the upstream changes between the two releases in order to determine which adjustments are likely to be required in the `build2` package. We can use the `upstream` submodule for that, which contains the change history we need.

One way to get an overview of changes between the releases is to use a graphical repository browser such as `gitk` and view a cumulative `diff` of changes between the two versions. For example, assuming the latest packaged version is tagged `v2.1.0` and the new version is tagged `v2.2.0`:

```
$ cd upstream/
$ gitk v2.1.0..v2.2.0 &
```

Then click on the commit tagged `v2.2.0`, scroll down and right-click on the commit tagged `v2.1.0`, and select the "Diff this -> selected" menu item. This will display the cumulative set of changes between these two upstream versions. Review them looking for the following types of changes in particular (discussed in the following sections):

- Changes to the source code layout.
- New dependencies being added or old removed.
- New source files being added or old removed (including in tests, etc).
- Changes to the upstream build system.
- Other new files/subdirectories being added or old removed.

2.11.7 New version: layout changes

As mentioned earlier, for drastic layout changes it may make sense to start from scratch and re-generate the package with the `bdep-new` command (use Decide on the package source code layout as a starting point). On the other hand, if the changes are minor, then you can try to adjust things manually. An in-between strategy is to generate the new layout using `bdep-new` on the side and then retrofit the relevant changes in `buildfiles` to the existing package. In a sense, this approach uses `bdep-new` as a guide to figure out how to implement the new layout.

2.11.8 New version: new/old dependencies

If upstream added new or removed old dependencies, then you will need to replicate these changes in your package as in the Add dependencies and Adjust source `buildfile`: dependencies initial packaging steps.

2.11.9 New version: new/old source files

If upstream added new or removed old source files, then you will need to replicate these changes in your package as in the Fill with upstream source code and possibly Adjust header `buildfile` and Adjust source `buildfile`: sources, private headers initial packaging steps.

Also don't forget about tests, examples, etc., which may also add new or remove old source files (typically new tests). See Convert smoke test to upstream tests.

If there are any manual modifications to the upstream source code, then you will also need to re-apply them to the new version as discussed in Modifying upstream source code manually.

2.11.10 New version: changes to build system

If upstream changed anything in the build system, then you may need to replicate these changes in your package's `buildfiles`. The relevant initial packaging steps are: Adjust project-wide build system files in `build/` and Adjust source `buildfile`: build and export options.

The corresponding steps for tests are: Adjust project-wide build system files in `tests/build/` and Convert smoke test to upstream tests.

2.11.11 New version: other new/old files/subdirectories

If upstream added or removed any other files or subdirectories that are relevant to our package (such as documentation), then adjust the package similar to the Adjust root `buildfile` and Adjust root `buildfile`: other subdirectories initial packaging steps.

2.11.12 New version: review `manifest` and `PACKAGE-README.md`

It makes sense to review the package `manifest` (Adjust `manifest`) and `PACKAGE-README.md` (Adjust `PACKAGE-README.md`) for any updates.

2.11.13 New version: review repository `README.md`

If any new packages were added in this version or if there are any changes to the development workflow, then it makes sense to review and if necessary update package repository `README.md` (Adjust package repository `README.md`).

2.11.14 New version: review/fix accumulated issues

When a bug is identified in an already released package version, we may not always be able to fix it immediately (for example, by releasing a revision). This could be because the change is too extensive/risky for a revision or simply not critical enough to warrant a release. In such cases it's recommended to file an issue in the package repository with the view to fix it when the next opportunity arises. Releasing a new upstream version is one such opportunity and it makes sense to review any accumulated package issues and see if any of them could be addressed.

2.11.15 New version: test locally and with CI

Once all the adjustments are in, test the package both locally and with CI similar to how we did it during the initial packaging after completing the smoke test:

Test locally
Test locally: installation
Test locally: distribution
Commit and test with CI

2.11.16 New version: merge, release, and publish

When the new version of the package is ready to be released, merge the work branch to `master` (or equivalent):

```
$ git checkout master
$ git merge --ff-only wip-2.2.0
```

Then release and publish using the same steps as after the initial packaging: Release and publish.

2.11.17 New version/revision in old release series

As discussed in Package version management, if we have already switched to the next upstream version in the `master` (or equivalent) branch, we cannot go back and release a new version or revision for an older release series on the same branch. Instead, we need to create a separate, long-lived branch for this work.

As an example, let's say we need to release another revision or a patch version for an already released `2.1.0` while our `master` branch has already moved on to `2.2.0`. In this case we create a new branch, called `2.1`, to continue with the `2.1.Z` release series. The starting point of this branch should be the latest released version/revision in the `2.1` series. Let's say in our case it is `2.1.0+2`, meaning we have released two revisions for `2.1.0` on the `master` branch before upgrading to `2.2.0`. Therefore we use the `v2.1.0+2` release tag to start the `2.1` branch:

```
$ git checkout -b 2.1 v2.1.0+2
```

Once this is done, we continue with the same steps as in New revision or New version except that we never merge this branch to `master`. If we ever need to release another revision or version in this release series, then we continue using this branch. In a sense, this branch becomes the equivalent of the `master` branch for this release series and you should treat it as such (once published, never delete, rewrite its history, etc).

It is less likely but possible that you may need to release a new minor version in an old release series. For example, the master branch may have moved on to `3.0.0` and you want to release `2.2.0` after the already released `2.1.0`. In this case it makes sense to call the branch `2` since it corresponds to the `2.Y.Z` release series. If you already have the `2.1` branch, then it makes sense to rename it to `2`.

3 What Not to Do

This chapter describes the common anti-patterns along with the recommended alternative approaches.

3.1 Don't write `buildfiles` from scratch, use `bdep-new`

Unless you have good reasons not to, create the initial project layout automatically using **`bdep-new(1)`**, then tweak it if necessary and fill with upstream source code.

The main rationale here is that there are many nuances in getting the build right and auto-generated `buildfiles` had years of refinement and fine-tuning. The familiar structure also makes it easier for others to understand your build, for example while reviewing your package submission or helping out with the package maintenance.

The **`bdep-new(1)`** command supports a wide variety of source layouts. While it may take a bit of time to understand the customization points necessary to achieve the desired layout for your first package, this will pay off in spades when you work on converting subsequent packages.

See Craft `bdep new` command line to create package for details.

3.2 Avoid fixing upstream issues in the `build2` package

Any deviations from upstream makes the `build2` package more difficult to maintain. In particular, if you make a large number of changes to the upstream source code, releasing a new version will require a lot of work. As a result, it is recommended to avoid fixing upstream issues in the `build2` package. Instead, try to have the issues fixed upstream and wait for them to be released as a new version.

Sometimes, however, you may have no choice. For example, upstream is inactive or has no plans to release a new version with your fixes any time soon. Or you may want to add support for a platform/compiler that upstream is not willing or capable of supporting.

Note that even if you do fix some issues in the `build2` package directly, try hard to also incorporate them upstream so that you don't need to maintain the patches forever.

See also Avoid changing upstream source code layout and How do I patch upstream source code?

3.3 Avoid changing upstream source code layout

It's a good idea to stay as close to the upstream's source code layout as possible. For background and rationale, see Decide on the package source code layout.

3.4 Don't make library header-only if it can be compiled

Some libraries offer two alternative modes: header-only and compiled. Unless there are good reasons not to, a `build2` build of such a library should use the compiled mode.

Some libraries use the *precompiled* term to describe the non-header-only mode. We don’t recommend using this term in the `build2` package since it has a strong association with precompiled headers and can therefore be confusing. Instead, use the *compiled* term.

The main rationale here is that a library would not be offering a compiled mode if there were no benefits (usually faster compile times of library consumers) and there is no reason not to take advantage of it in the `build2` package.

There are, however, valid reasons why a compiled mode cannot be used, the most common of which are:

- The compiled mode is not well maintained/tested by upstream and therefore offers inferior user experience.
- The compiled mode does not work on some platforms, usually Windows due to the lack of symbol export support (but see Automatic DLL Symbol Exporting).
- Uses of the compiled version of the library requires changes to the library consumers, for example, inclusion of different headers.

If a compiled mode cannot always be used, then it may be tempting to support both modes by making the mode user-configurable. Unless there are strong reasons to, you should resist this temptation and, if the compiled mode is not universally usable, then use the header-only mode everywhere.

The main rationale here is that variability adds complexity which makes the result more prone to bugs, more difficult to use, and harder to review and maintain. If you really want to have the compiled mode, then the right way to achieve it is to work with upstream to fix any issues that prevent its use in `build2`.

There are, however, valid reasons why supporting both modes may be needed, the most common of which are:

- The library is widely used in both modes but switching from one mode to the other requires changes to the library consumers (for example, inclusion of different headers). In this case only supporting one mode would mean not supporting a large number of library consumers.
- The library consists of a large number of independent components while its common for applications to only use a small subset of them. And compiling all of them in the compiled mode takes a substantial amount of time. Note that this can alternatively be addressed by making the presence of optional components user-configurable.

3.5 Don’t bundle dependencies

Sometimes third-party projects bundle their dependencies with their source code (also called vendoring). For example, a C++ library may bundle a testing framework. This is especially common with `catch2` where one often encounters a comical situation with only a few kilobytes of library source code and over 600KB of `catch2.hpp`.

The extra size, while wasteful, is not the main issue, however. The bigger problem is that if a bug is fixed in the bundled dependency, then to propagate the fix we will need to release a new version (or revision) of each package that bundles it. Needless to say this is not scalable.

While this doesn't apply to testing frameworks, an even bigger issue with bundling of dependencies in general is that two libraries that bundle the same dependency (potentially of different versions) may not be able to coexist in the same build with the symptoms ranging from compile errors to subtle runtime issues that are hard to diagnose.

As a result, it is strongly recommended that you unbundle any dependencies that upstream may have bundled. In case of testing frameworks, see How do I handle tests that have extra dependencies? for the recommended way to deal with such cases.

One special case where a bundled dependency may be warranted is a small utility that is completely inline/private to the implementation and where making it an external dependency may lead to a less performant result (due to the inability to inline without resorting to LTO). The `xxhash` implementation in `libzstd` is a representative example of this situation.

3.6 Don't build your main targets in the root `buildfile`

It may be tempting to have your main targets (libraries, executables) in the root `buildfile`, especially if it allows you to symlink entire directories from `upstream/` (which is not possible if you have to have a `buildfile` inside). However, this is not recommended except for the simplest of projects.

Firstly, this quickly gets messy since you have to combine managing `README`, `LICENSE`, etc., and subdirectories with your main target builds. More importantly, this also means that when your main target is imported (and thus the `buildfile` that defines this target must be loaded), your entire project will be loaded, including any `tests/` and `examples/` subprojects, and that is wasteful.

If you want to continue symlinking entire directories from `upstream/` but without moving everything to the root `buildfile`, the recommended approach is to simply add another subdirectory level. Let's look at a few concrete example to illustrate the technique (see Decide on the package source code layout for background on the terminology used).

Here is the directory structure of a package which uses a combined layout (no header/source split) and where the library target is in the root `buildfile`:

```
libigl-core/
|-- igl/ -> ../upstream/igl/
|-- tests/
`-- buildfile                          # Defines lib{igl-core}.
```

And here is the alternative structure where we have added the extra `libigl-core` subdirectory with its own `buildfile`:

```
libigl-core/
|-- libigl-core/
|Â Â  |-- igl/ -> ../../upstream/igl/
|Â Â  `-- buildfile                       # Defines lib{igl-core}.
|-- tests/
`-- buildfile
```

Below is the bdep-new invocation that can be used to automatically create this alternative structure (see Craft bdep new command line to create package for background and **bdep-new(1)** for details):

```
$ bdep new \
 --type lib,prefix=libigl-core,subdir=igl,buildfile-in-prefix \
 libigl-core
```

Let’s also look at an example of the split layout, which may require a slightly different bdep-new sub-options to achieve the same result. Here is the layout which matched upstream exactly:

```
$ bdep new --type lib,split,subdir=foo,no-subdir-source libfoo
$ tree libfoo
libfoo/
|-- include/
|Â Â  `-- foo/
|Â Â      |-- buildfile
|Â Â      `-- ...
`-- src/
    |-- buildfile
    `-- ...
```

However, with this layout we will not be able to symlink the entire include/foo/ and src/ subdirectories because there are buildfiles inside (and which may tempt you to just move everything to the root buildfile). To fix this we can move the buildfiles out of source subdirectory foo/ and into prefixes (include/ and src/) using the buildfile-in-prefix sub-option. And since src/ doesn’t have a source subdirectory, we have to invent one:

```
$ bdep new --type lib,split,subdir=foo,buildfile-in-prefix libfoo
$ tree libfoo
libfoo/
|-- include/
|Â Â  |-- foo/ -> ../../upstream/include/foo/
|Â Â  `-- buildfile
`-- src/
    |-- foo/ -> ../../upstream/src/
    `-- buildfile
```

3.7 Don’t make extensive changes in a revision

Unlike a new version, a revision replaces the previous revision of the same version and as a result must be strictly backwards-compatible in all aspects with what it replaces. If you make extensive changes in a revision, it becomes difficult to guarantee that this requirement is satisfied. As a result, you should refrain from making major changes, like substantially altering the build, in a revision, instead delaying such changes until the next version.

4 Packaging HOWTO

This chapter provides advice on how to handle less common packaging tasks and requirements.

4.1 How do I patch upstream source code?

If you need to change something in the upstream source code, there are several options: You can make a copy of the upstream source file and make the modifications there. While straightforward, this approach has one major drawback: you will have to keep re-applying the changes for every subsequent version unless and until upstream incorporates your changes. The other two options try to work around this drawback.

The first alternative option is to modify the upstream source code automatically during the build, typically using an ad hoc recipe. This approach works best when the changes are regular and can be applied mechanically with something like the `sed` builtin.

The second alternative option is to use the C/C++ preprocessor to make the necessary changes to the upstream source code during compilation. Unlike the first alternative, this approach doesn't have a prescribed way to apply it in every situation and often requires some imagination. Note that it also has the tendency to quickly get out of hand, at which point it's wise to keep it simple and use the first option (manual modification).

The following sections examine each approach in detail.

4.1.1 Modifying upstream source code manually

As an illustration of this approach, let's say we need to patch `src/foo.cpp` in our `libfoo` example from the previous sections (see the Fill with upstream source code step for a refresher). The recommended sequence of steps is as follows:

1. Rename the upstream symlink to `.orig`:

   ```
   $ cd libfoo/src/
   $ mv foo.cpp foo.cpp.orig
   ```

2. Make a deep copy of `.orig`:

   ```
   $ cp -H foo.cpp.orig foo.cpp
   ```

3. Make any necessary modifications in the deep copy:

   ```
   $ edit foo.cpp
   ```

4. Create a patch for the modifications:

   ```
   $ diff -u foo.cpp.orig foo.cpp >foo.cpp.patch
   ```

The presence of the `.orig` and `.patch` files makes it clear that the upstream code was modified. They are also useful when re-applying the changes to the new version of the upstream source code. The recommended sequence of steps for this task is as follows:

1. After the `upstream` submodule update (see the New version: update `upstream` submodule step), the `.orig` symlink points to the new version of the upstream source file.
2. Overwrite old modified version with a deep copy of new `.orig`:

   ```
   $ cp -H foo.cpp.orig foo.cpp
   ```

3. Apply old modifications to the new deep copy:

   ```
   $ patch <foo.cpp.patch
   ```

   If some hunks of the patch could not be applied, manually resolve any conflicts.

4. If in the previous step the patch did not apply cleanly, regenerate it:

   ```
   $ diff -u foo.cpp.orig foo.cpp >foo.cpp.patch
   ```

4.1.2 Modifying upstream source code during build

As an illustration of this approach, let's say upstream is using the `${VAR}` style variable substitutions in their `config.h.cmake` instead of the more traditional `@VAR@` style:

```
/* config.h.cmake */

#define FOO_VERSION "${PROJECT_VERSION}"
```

The `${VAR}` style is not supported by the `build2 autoconf` module which means we cannot use the upstream `config.h.cmake` as is. While we could patch this file manually to use `@VAR@` instead, this is a pretty mechanical change that can be easily made with a simple ad hoc recipe during the build, freeing us from manually applying the same changes in subsequent versions. For example:

```
using autoconf

h{config}: in{config.h.in}
{
  autoconf.flavor = cmake
  PROJECT_VERSION = $version
}

in{config.h.in}: file{config.h.cmake}
{{
  sed -e 's/\$\{(.+)\}/@\1@/g' $path($<) >$path($>)
}}
```

4.1.3 Modifying upstream source code with C/C++ preprocessor

A good illustration of this approach is adding the `build2` metadata to an executable (see How do I convey additional information (metadata) with executables and C/C++ libraries? for background). Let's say we have a symlink to upstream's `main.c` that implements the executable's `main()` function and we need to add a snipped of code at the beginning of this function that handles the `--build2-metadata` option. While manually modifying `main.c` is not a wrong approach, we can try to be clever and do it automatically with the preprocessor.

Specifically, we can create another file next to the `main.c` symlink, calling it, for example, `main-build2.c`, with the following contents:

```
/* Handle --build2-metadata in main() (see also buildfile). */

#define main xmain
#include "main.c"
#undef main

#include <stdio.h>
#include <string.h>

int main (int argc, const char** argv)
{
  if (argc == 2 && strncmp (argv[1], "--build2-metadata=", 18) == 0)
  {
    printf ("# build2 buildfile foo\n");
    printf ("export.metadata = 1 foo\n");
    printf ("foo.name = [string] foo\n");
    ...
    return 0;
  }

  return xmain (argc, argv);
}
```

The idea here is to rename the original `main()` with the help of the C preprocessor and provide our own `main()` which, after handling `--build2-metadata` calls the original. One notable deal-breaker for this approach would be a C++ implementation of `main()` that doesn't have the explicit `return`. There is also a better chance in C++ for the `main` macro to replace something unintended.

To complete this we also need to modify our `buildfile` to exclude `main.c` from compilation (since it is compiled as part of `main-build2.c` via the preprocessor inclusion). For example:

```
exe{foo}: {h c}{** -main}
exe{foo}: c{main}: include = adhoc  # Included in main-build2.c.
```

4.2 How do I deal with bad header inclusion practice?

This section explains how to deal with libraries that include their public, generically-named headers without the library name as a subdirectory prefix. Such libraries cannot coexist, neither in the same build nor when installed.

Specifically, as an illustration of the problem, consider the `libfoo` library with a public header named `util.h` that is included as `<util.h>` (instead of, say, `<libfoo/util.h>` or `<foo/util.h>`). If this library's headers are installed directly into, say, `/usr/include`, then if two such libraries happened to be installed at the same time, then one will overwrite the other's header. There are also problems in the non-installed case: if two such libraries are used by the same project, then which `<util.h>` header gets included depends on which library's header search path ends up being specified first on the command line (with the `-I` option).

These issues are severe enough that libraries with such inclusion issues cannot be published to cppget.org without them being addressed in the `build2` package. Thankfully, most library authors these days use the library name as an inclusion prefix (or sometimes they have headers that are decorated with the library name). However, libraries that do not follow these guidelines do exist and this section describes how to change their inclusion scheme if you are attempting to package one of them.

One notable consequence of changing the inclusion scheme is that it will no longer be possible to use a system-installed version of the package (because it presumably still uses the unqualified inclusion scheme). Note, however, that distributions like Debian and Fedora have the same co-existence issue as we do and are generally strict about potential header clashes. In particular, it is not uncommon to find Debian packages installing library headers into subdirectories of `/usr/include` to avoid such clashes. And if you find this to be the case for the library you are packaging, then it may make sense to use the same prefix as used by the main distributions for compatibility.

It is also possible that distributions disregard these considerations for some libraries. This usually happens for older, well-known libraries that happened to be installed this way in the early days and changing things now will be too disruptive. In a sense, it is understood that such libraries effectively "own" the unqualified header names that they happen to be using. If you think you are packaging such a library, get in touch to discuss this further since it may make sense to also disregard this rule in cppget.org.

As a concrete example of the approach, let's continue with `libfoo` that has `util.h` and which upstream expects the users to include as `<util.h>`. The is what the upstream source code layout may look like:

```
libfoo/
|-- include/
|Â Â  `-- util.h
`-- src/
    `-- ...
```

Our plan is to change the inclusion scheme in the build2 package from <util.h> to <libfoo/util.h>. To this effect, we use a slightly modified layout for our package (see Craft bdep new command line to create package on how to achieve it):

```
libfoo/
|-- include/
|Â Â  ·-- libfoo/
|Â Â      ·-- util.h  -> ../../../upstream/include/util.h
·-- src/
    ·-- ...          -> ../../upstream/src/...
```

The installation-related section in our header buildfile will look like this:

```
# Install into the libfoo/ subdirectory of, say, /usr/include/
# recreating subdirectories.
#
{hxx ixx txx}{*}:
{
  install         = include/libfoo/
  install.subdirs = true
}
```

In the source buildfile we will most likely need to add the include/libfoo header search path since the upstream source files continue to include public headers without the library prefix (there should be no harm in that and it's not worth modifying them):

```
# Build options.
#
out_pfx_inc = [dir_path] $out_root/include/
src_pfx_inc = [dir_path] $src_root/include/
out_pfx_src = [dir_path] $out_root/src/
src_pfx_src = [dir_path] $src_root/src/

# Unqualified (without <libfoo/...>) header search paths.
#
out_pfx_inc_unq = [dir_path] $out_root/include/libfoo
src_pfx_inc_unq = [dir_path] $src_root/include/libfoo

cxx.poptions =+ "-I$out_pfx_src" "-I$src_pfx_src" \
                "-I$out_pfx_inc" "-I$src_pfx_inc" \
                "-I$out_pfx_inc_unq" "-I$src_pfx_inc_unq"
```

It is also possible that public headers include each other as <util.h> rather than the more common "util.h". If that's the case, then we need to fix that and there are two ways to do it. The first approach is to patch the public headers to include each other with the library prefix (that is, <libfoo/util.h>, etc). See How do I patch upstream source code? for details.

The second approach is to support including public headers both ways, that is, as <libfoo/util.h> and as <util.h>. This will not only solve the above problem (public headers including each other), but also support any existing code that uses this library and most likely includes its headers the old way, without the prefix.

There is, however, a major drawback to doing that: while the installation of the library can now co-exist with other libraries (because we install its public headers into, say, `/usr/include/libfoo`), it may still not be usable in combination with other libraries from the same build (because we still add the unqualified header search path).

If you still want to provide this dual inclusion support, the way to achieve it is by exporting the unqualified header search path and also adding it to the `pkg-config` files (see How do I handle extra header installation subdirectory? for background on the latter). For example:

```
# Export options.
#
lib{foo}:
{
  cxx.export.poptions = "-I$out_pfx_inc" "-I$src_pfx_inc" \
                        "-I$out_pfx_inc_unq" "-I$src_pfx_inc_unq"
  cxx.export.libs = $intf_libs
}

# Make sure headers installed into, say, /usr/include/libfoo/
# can also be included without the directory prefix for backwards
# compatibility.
#
lib{foo}: cxx.pkgconfig.include = include/ include/libfoo/
```

4.3 How do I handle extra header installation subdirectory?

This section explains how to handle an additional header installation subdirectory. As an illustration of the problem, consider the `libfoo` example from the previous sections (see the Fill with upstream source code step for a refresher). In that example the library headers are included as `<foo/util.hpp>` and installed as, say, `/usr/include/foo/util.hpp`. In this scheme the installed header inclusion works without requiring any extra steps from our side because the compiler searches for header in `/usr/include` by default.

However, some libraries choose to install their headers into a subdirectory of, say, `/usr/include` but without having this subdirectory as part of the inclusion path (`foo/` in `<foo/util.hpp>`). The two typical reasons for this are support for installing multiple versions of the same library side-by-side (for example, `/usr/include/foo-v1/foo/util.hpp`) as well as not using the library name as the inclusion subdirectory prefix and then having to hide the headers in a subdirectory due to potential clashes with other headers (if installed directly into, say, `/usr/include`; see How do I deal with bad header inclusion practice? for background).

In such cases the installed header inclusion does not work out of the box and we have to arrange for an additional header search path to be added via `pkg-config`. Let's use the versioned library case to illustrate this technique. The relevant part from the header `buildfile` will look like this:

```
# Install into the foo-vN/foo/ subdirectory of, say, /usr/include/
# recreating subdirectories.
#
{hxx ixx txx}{*}:
{
  install         = include/"foo-v$version.major"/foo/
  install.subdirs = true
}
```

The part that we need to add, this time to the source `buildfile`, looks like this:

```
# Make sure headers installed into, say, /usr/include/foo-vN/foo/
# can be included as <foo/*.hpp> by overriding the header search
# path in the generated pkg-config files.
#
lib{foo}: cxx.pkgconfig.include = include/"foo-v$version.major"/
```

The variable will be `c.pkgconfig.include` for a C library.

4.4 How do I handle headers without an extension?

If all the headers in a project have no extension, then you can simply specify the empty `extension` value for the `hxx{}` target type in `build/root.build`:

```
hxx{*}: extension =
cxx{*}: extension = cpp
```

Note, however, that using wildcard patterns for such headers in your `buildfile` is a bad idea since such a wildcard will most likely pick up other files that also have no extension (such as `buildfile`, executables on UNIX-like systems, etc). Instead, it's best to spell the names of such headers explicitly. For example, instead of:

```
lib{hello}: {hxx cxx}{*}
```

Write:

```
lib{hello}: cxx{*} hxx{hello}
```

If only some headers in a project have no extension, then it's best to specify the non-empty extension for the `extension` variable in `build/root.build` (so that you can still use wildcards for headers with extensions) and spell out the headers with no extension explicitly. Continuing with the above example, if we have both the `hello.hpp` and `hello` headers, then we can handle them like this:

```
hxx{*}: extension = hpp
cxx{*}: extension = cpp

lib{hello}: {hxx cxx}{*} hxx{hello.}
```

Notice the trailing dot in `hxx{hello.}` – this is the explicit "no extension" specification. See Targets and Target Types for details.

4.5 How do I expose extra debug macros of a library?

Sometimes libraries provide extra debugging facilities that are usually enabled or disabled with a macro. For example, `libfoo` may provide the `LIBFOO_DEBUG` macro that enables additional sanity checks, tracing, etc. Normally, such facilities are disabled by default.

While it may seem like a good idea to detect a debug build and enable this automatically, it is not: such facilities usually impose substantial overheads and the presence of debug information does not mean that performance is not important (people routinely make optimized builds with debug information).

As a result, the recommended approach is to expose this as a configuration variable that the consumers of the library can use (see Project Configuration for background). Continue with the `libfoo` example, we can add `config.libfoo.debug` to its `build/root.build`:

```
# build/root.build

config [bool] config.libfoo.debug ?= false
```

And then define the `LIBFOO_DEBUG` macro based on that in the `buildfile`:

```
# src/buildfile

if $config.libfoo.debug
  cxx.poptions += -DLIBFOO_DEBUG
```

If the macro is also used in the library's interface (for example, in inline or template functions), then we will also need to export it (see Adjust source `buildfile`: build and export options for details):

```
# src/buildfile

if $config.libfoo.debug
{
  cxx.poptions += -DLIBFOO_DEBUG
  lib{foo}: cxx.export.poptions += -DLIBFOO_DEBUG
}
```

If the debug facility in question should be enabled by default even in the optimized builds (in which case the macro usually has the `NO_DEBUG` semantics), the other option is to hook it up to the standard `NDEBUG` macro, for example, in the library's configuration header file.

Note that such `.debug` configuration variables should primarily be meant for the user to selectively enable extra debugging support in certain libraries of their build. However, if your project depends on a number of libraries with such extra debugging support and it generally makes sense to also enable this support in dependencies if it is enabled in your project, then you may want to propagate your `.debug` configuration value to the dependencies (see the `depends` package `manifest` value for details on dependency configuration). You, however, should still allow the user to override this decision on the per-dependency basis.

Continuing with the above example, let's say we have `libbar` with `config.libbar.debug` that depends on `libfoo` and wishes by default to enable debugging in `libfoo` if it is enabled in `libbar`. This is how we can correctly arrange for this in `libbar`'s `manifest`:

```
depends:
\
libfoo ^1.2.3
{
  # We prefer to enable debug in libfoo if enabled in libbar
  # but accept if it's disabled (for example, by the user).
  #
  prefer
  {
    if $config.libbar.debug
      config.libfoo.debug = true
  }

  accept (true)
}
\
```

5 Packaging FAQ

5.1 Publishing FAQ

5.1.1 Why is my package in `alpha` rather than `stable`?

If your package uses a semver version (or semver-like, that is, has three version components) and the first component is zero (for example, `0.1.0`), then, according to the semver specification, this is an alpha version and **`bdep-publish(1)`** automatically published such a version to the `alpha` section of the repository.

Sometimes, however, in a third-party package, while the version may look like semver, upstream may not assign the zero first component any special meaning. In such cases you can override the `bdep-publish` behavior with the `--section` option, for example:

```
$ bdep publish --section=stable
```

Note that you should only do this if you are satisfied that by having the zero first component upstream does not imply alpha quality. Getting an explicit statement to this effect from upstream is recommended.

5.1.2 Where to publish if package requires staged toolchain?

If your package requires the staged toolchain, for example, because it needs a feature or bugfix that is not yet available in the released toolchain, then you won't be able to publish it to `cppget.org`. Specifically, if your package has the accurate `build2` version constraint and you attempt to publish it, you will get an error like this:

```
error: package archive is not valid
  info: unable to satisfy constraint (build2 >= 0.17.0-) for package foo
  info: available build2 version is 0.16.0
```

There are three alternative ways to proceed in this situation:

1. Wait until the next release and then publish the package to `cppget.org`.
2. If the requirement for the staged toolchain is "minor", that is, it doesn't affect the common functionality of the package or only affects a small subset of platforms/compilers, then you can lower the toolchain version requirement and publish the package to `cppget.org`. For example, if you require the staged toolchain because of a bugfix that only affects one platform, it doesn't make sense to delay publishing the package since it is perfectly usable on all the other platforms in the meantime.
3. Publish it to queue.stage.build2.org, the staging package repository. This repository contain new packages that require the staged toolchain to work and which will be automatically moved to `cppget.org` once the staged version is released. The other advantage of publishing to this repository (besides not having to remember to manually publish the package once the staged version is released) is that your package becomes available from an archive repository, which is substantially faster than a `git` repository.

   To publish to this repository, use the following `bdep-publish` command line:

   ```
   $ bdep publish --repository=https://stage.build2.org ...
   ```

5.1.3 Why "project owner authentication failed" while publishing?

If you are getting the following error while attempting to publish a new version of a package:

```
$ bdep publish
...
error: project owner authentication failed
```

Then this means the remote `git` repository you are using does not match the one from which you (or someone else) has published the initial version of the package.

In `build2` we use the ownership of the package `git` repository as a proxy for the ownership of the package name on cppget.org. Specifically, when you publish the package for the first time, we record the `git` URL for its package repository. And any further versions of this package can only be submitted by someone who has write access to this repository. See **`bdep-publish(1)`** for details.

Based on this background, the first step you need to take when getting the above owner authentication error is to understand its cause. For that, first use the `git-config` command to see the URL you are using locally:

```
$ git config --get remote.origin.url
```

Then look in the `git` repositories that back cppget.org and queue.cppget.org and find the URL that is recorded in the `owners/` subdirectory in the corresponding `package-owner.manifest` file.

Note that your local URL will normally be SSH while the recorded URL will always be HTTPS. Provided that the host names match, the part to look in for differences is the path component. One common cause of a mismatch is the missing `.git` extension. For example (local first, recorded second):

```
git@github.com:build2-packaging/zstd
https://github.com/build2-packaging/zstd.git
```

In this case adding the missing extension to the local URL should fix the error.

If, however, the discrepancy is expected, for example, because you have renamed the package repository or moved it to a new location, the ownership information will need to be updated manually. In this case feel free to submit a pull request with the necessary changes or get in touch.