# The `build2` Toolchain Installation and Upgrade



Revision `0.16`, June 2023
This revision of the document describes the `build2` toolchain `0.16.x` series.

# Table of Contents

# Preface

The recommended way to install and upgrade the `build2` toolchain in most circumstances is using the automated install scripts as described on the Install page. This document describes the manual installation and upgrade process which offers greater control that can be helpful in situations where the automated scripts cannot be used (packaging for system package managers, unsupported by the install scripts platform/compiler, etc).

# 1 Introduction

One of the primary goals of the `build2` toolchain is to provide a uniform build interface across all the platforms and compilers. As a result, if you already have the toolchain installed and would like to upgrade to a newer version, then there is a single set of upgrade instructions for all the platforms.

If, however, you need to install the toolchain for the first time, then it has to be bootstrapped and that process is platform-specific. The rest of this section discusses a few general bootstrap considerations and then directs you to the appropriate platform-specific instructions.

In the rest of this guide we use the `$` symbol for a UNIX shell prompt and > for the Windows command prompt. Similarly, we use `\` for UNIX command line continuations and `^` for Windows. Usually you should be able to copy and paste (sans the prompt) example commands in order to execute them but sometimes you might need to change a thing or two (for example, replace `X.Y.Z` with the actual version). Once we are able to use the `build2` toolchain, the command line interface becomes regular and we usually only show the UNIX version of the commands. In this case making a Windows version is a simple matter of adjusting paths and, if used, line continuations.

The `build2` toolchain requires a C++14 compiler. From the commonly-used options, GCC 4.9, Clang 3.7, and MSVC 14 (2015) Update 3 or any later versions of these compilers should work.

Note that the C++ compiler that you use to build the `build2` toolchain and the one that you will use to build your projects need not be the same. For example, if you are using MSVC 12 (2013) (which cannot build `build2`), it is perfectly fine to get a minimal MinGW toolchain and use that to build `build2`; you will still be able to use MSVC 12 to build your own code.

At the high level, the bootstrap process involves the following 5 steps.

**1. Bootstrap, Phase 1**

First, a minimal build system executable is built using provided shell scripts/batch files or a GNU makefile. The result is only guaranteed to be able to rebuild the build system itself.

**2. Bootstrap, Phase 2**
Then, the build system is rebuilt with static libraries. The result is only guaranteed to be able to build the build system and the package manager.

**3. Stage**
At this step the build system and package manager are built with shared libraries and then staged.

**4. Install**
Next, the staged tools are used to build and install the entire toolchain from the package repository with the package manager.

**5. Clean**
Finally, the staged at step 3 tools are uninstalled.

The end result of the bootstrap process is the installed toolchain as well as the package manager configuration (created at step 4) that can be used to upgrade to newer versions.

You can skip step 4 and instead install at step 3 if for some reason you prefer not to build from packages (for example, because the machine is offline).

For Windows, if you are using either MSVC, Clang, or MinGW, continue with Bootstrapping on Windows. If using WSL, MSYS, or Cygwin, then instead refer to Bootstrapping on UNIX.

For Mac OS X, continue with Bootstrapping on Mac OS X.

For other UNIX-like operating systems (GNU/Linux, FreeBSD, etc; this also includes WSL, MSYS, and Cygwin), continue with Bootstrapping on UNIX.

# 2 Bootstrapping on Windows

The following instructions are for bootstrapping `build2` from the Windows command prompt with either MSVC, Clang (targeting the MSVC runtime), or MinGW. If you are using any kind of UNIX emulation layer (for example, WSL, MSYS, or Cygwin) and already have a UNIX shell with standard utilities, then you most likely should follow Bootstrapping on UNIX instead.

Note that if you continue with these instructions but you already have your own installation of MSYS and/or MinGW, then make sure that their paths are not in your `PATH` environment variable when building and using `build2` since they may provide conflicting DLLs.

The `build2` toolchain on Windows requires a set of extra utilities (`install`, `diff`, `curl`, `tar`, etc). These are provided by the `build2-baseutils` package (see the `README` file inside for details). Normally, the `build2` toolchain itself is installed into the same directory as the utilities in order to produce the combined installation.

To build on Windows you will need either MSVC 14 Update 3 or later, Clang 8 or later (either bundled with MSVC or installed separately), or MinGW GCC 4.9 or later. If you don't already have a suitable C++ compiler, then you can use the `build2-mingw` package which provides a minimal MinGW-W64 GCC distribution (see the `README` file inside for details). If used, then it should be unpacked into the same directory as `build2-baseutils`.

If using your own MinGW GCC installation, make sure it is configured with the `posix` threading model (this is currently the only configuration that implements C++11 threads; run `g++ -v` to verify).

Only 64-bit variants of the `baseutils` and `mingw` packages are provided and they **must** match the width of the `build2` toolchain. Note also that the 64-bit `build2` toolchain can be used to build 32-bit applications without any restrictions.

To bootstrap on Windows with either MSVC, Clang, or MinGW start with the following common steps:

**1. Open Command Prompt**

Start the standard Windows Command Prompt. If you plan to build with MSVC or Clang bundled with MSVC, then you may go ahead and start the Visual Studio "x64 Native Tools Command Prompt" (or wait for MSVC/Clang-specific instructions).

**2. Create Build Directory**

You will want to keep this directory around in order to upgrade to new toolchain versions in the future. In this guide we use `C:\build2-build\` as the build directory and `C:\build2\` as the installation directory but you can use other paths.

```
> cd /D C:\
> mkdir build2-build
> cd build2-build
```

**3. Download Archives**

Download the following files as well as their `.sha256` checksums from the Download page:

```
build2-baseutils-X.Y.Z-x86_64-windows.zip
build2-mingw-X.Y.Z-x86_64-windows.tar.xz   (if required)
build2-toolchain-X.Y.Z.tar.xz
```

Place everything into `C:\build2-build\` (build directory).

**4. Verify Archive Checksums**

Verify archive checksums match (compare visually):

```
> type *.sha256
> for %f in (*.zip *.xz) do certutil -hashfile %f SHA256
```

**5. Unpack `build2-baseutils`**

Unpack the `build2-baseutils-X.Y.Z-x86_64-windows.zip` archive into `C:\` using Windows Explorer (for example, copy the archive directory and then paste it). Rename it to `C:\build2\`. This will be the toolchain installation directory.

**6. Set `PATH`**

Set the `PATH` environment variable and verify that the utilities are found and work:

```
> set "PATH=C:\build2\bin;%PATH%"
> where tar
> tar --version
```

**7. Unpack `build2-mingw` (optional)**

If required, unpack the `build2-mingw-X.Y.Z-x86_64-windows.tar.xz` archive into `C:\build2\`:

```
> xz -d build2-mingw-X.Y.Z-x86_64-windows.tar.xz
> tar -xf build2-mingw-X.Y.Z-x86_64-windows.tar ^
  --one-top-level=C:\build2 --strip-components=1
```

Verify that the MinGW GCC is found and works:

```
> where g++
> g++ --version
```

**8. Unpack `build2-toolchain`**

Unpack the `build2-toolchain-X.Y.Z.tar.xz` archive and change to its directory:

```
> xz -d build2-toolchain-X.Y.Z.tar.xz
> tar -xf build2-toolchain-X.Y.Z.tar
> cd build2-toolchain-X.Y.Z
```

If building with MSVC, continue with Bootstrapping on Windows with MSVC.

If building with Clang, continue with Bootstrapping on Windows with Clang.

If building with MinGW, continue with Bootstrapping on Windows with MinGW.

# 2.1 Bootstrapping on Windows with MSVC

Continuing from Bootstrapping on Windows, if you have already started an appropriate Visual Studio command prompt, then you can continue using it. Otherwise, start the Visual Studio "x64 Native Tools Command Prompt". Also set the `PATH` environment variable as on the previous steps:

```
> set "PATH=C:\build2\bin;%PATH%"
```

To build with MSVC you can either perform the following steps manually or, if after reviewing the steps you are happy with using the defaults, run the `build-msvc.bat` batch file. It performs (and echoes) the same set of steps as outlined below but only allows you to customize the installation directory and a few other things (run `build-msvc.bat /?` for usage).

For example, you could run this batch file (from the above-mentioned command prompt) like this:

```
> .\build-msvc.bat
```

Note that at about half way through (`bpkg fetch` at step 4 below) the script will stop and prompt you to verify the authenticity of the repository certificate. To run the script unattended you can specify the certificate fingerprint with the `--trust` option (see `build-msvc.bat /?` for details).

The end result of the bootstrap process (performed either with the script or manually) is the installed toolchain as well as the `bpkg` configuration in `build2-toolchain-X.Y\` that can be used to upgrade to newer versions. It can also be used to uninstall the toolchain:

```
> cd build2-toolchain-X.Y
> bpkg uninstall --all
```

Note that in both cases (manual or scripted bootstrap), if something goes wrong and you need to restart the process, you **must** start with a clean toolchain source by unpacking it afresh from the archive.

The rest of this section outlines the manual bootstrap process.

**1. Bootstrap, Phase 1**

First, we build a minimal build system with the provided `bootstrap-msvc.bat` batch file. Normally, the only argument you will pass to this script is the C++ compiler to use but there is also a way to specify compile options; run `bootstrap-msvc.bat /?` and see the `build2\INSTALL` file for details.

```
> cd build2
> .\bootstrap-msvc.bat cl /w /MP8

> build2\b-boot --version
```

**2. Bootstrap, Phase 2**

Then, we rebuild the build system with the result of Phase 1 linking libraries statically.

```
> build2\b-boot config.cxx=cl config.bin.lib=static build2\exe{b}
> move /y build2\b.exe build2\b-boot.exe

> build2\b-boot --version
```

**3. Stage**

At this step the build system and package manager are built with shared libraries and then staged:

```
> cd ..  # Back to build2-toolchain-X.Y.Z\

> build2\build2\b-boot configure      ^
  config.cxx=cl                       ^
  config.bin.lib=shared               ^
  config.bin.suffix=-stage            ^
  config.install.root=C:\build2       ^
  config.install.data_root=root\stage

> build2\build2\b-boot install: build2\ bpkg\
```

The strange-looking `config.install.data_root=root\stage` means install data files (as opposed to executable files) into the `stage\` subdirectory of wherever `config.install.root` points to (so in our case it will be `C:\build2\stage\`). This subdirectory is temporary and will be removed in a few steps.

Verify that the toolchain binaries can be found and work (this relies on the `PATH` environment variable we have set earlier):

```
> where b-stage
C:\build2\bin\b-stage.exe

> where bpkg-stage
C:\build2\bin\bpkg-stage.exe

> b-stage --version
> bpkg-stage --version
```

At the next step we will use `bpkg` to build and install the entire toolchain. If for some reason you prefer not to build from packages (for example, because the machine is offline), then you can convert this step into a local installation and skip the rest of the steps.

To perform a local installation with the `build-msvc.bat` batch file, pass the `--local` option.

To perform a local installation you will need to change the `configure` and `install` command lines above along these lines (see also a note on the following step about only building shared libraries, toolchain executables prefix/suffix, etc):

```
> build2\build2\b-boot configure ^
  config.config.hermetic=true    ^
  config.cxx=cl                  ^
  config.cc.coptions=/O2         ^
  config.bin.lib=shared          ^
  config.install.root=C:\build2

> build2\build2\b-boot install: build2\ bpkg\ bdep\
```

You will also need to build and install the standard build system modules:

```
> b install: !config.install.scope=project libbuild2-*\
```

To verify the build system modules installation you can load them with the following command:

```
> b noop: tests\libbuild2-*\
```

To uninstall such a local installation, run:

```
> b uninstall: build2\ bpkg\ bdep\ libbuild2-*\
```

**4. Install**

Next, we use the staged tools to build and install the entire toolchain from the package repository with the `bpkg` package manager. First, we create the `bpkg` configuration. The configuration values are pretty similar to the previous step and you may want/need to make similar adjustments.

```
> cd ..  # Back to build2-build\
> md build2-toolchain-X.Y
> cd build2-toolchain-X.Y

> bpkg-stage create             ^
  cc                            ^
  config.config.hermetic=true   ^
  config.cxx=cl                 ^
  config.cc.coptions=/O2        ^
  config.bin.lib=shared         ^
  config.install.root=C:\build2
```

The above configuration will only build shared libraries. If you would like to build both shared and static, remove `config.bin.lib=shared`.

To add a custom prefix/suffix to the toolchain executables names, add `config.bin.exe.prefix=...` and/or `config.bin.exe.suffix=...`.

The `config.config.hermetic=true` configuration variable in the above command makes sure the embedded `~host` and `~build2` configurations include the current environment. This is especially important for `~build2` which is used to dynamically build and load ad hoc recipes and build system modules and must therefore match the environment

that was used to build the build system itself.

Next, we add the package repository, build, and install:

```
> bpkg-stage add https://pkg.cppget.org/1/alpha
> bpkg-stage fetch
> bpkg-stage build --for install build2 bpkg bdep
> bpkg-stage install --all
```

By default `bpkg` will build the latest available version of each package. You can, however, specify the desired versions explicitly, for example:

```
> bpkg-stage build --for install build2/X.Y.Z bpkg/X.Y.Z bdep/X.Y.Z
```

To verify the result, run:

```
> where b
C:\build2\bin\b.exe

> where bpkg
C:\build2\bin\bpkg.exe

> where bdep
C:\build2\bin\bdep.exe

> b --version
> bpkg --version
> bdep --version
```

Finally, we build and install the standard build system modules:

```
> bpkg build --for install libbuild2-autoconf libbuild2-kconfig
> bpkg install !config.install.scope=project ^
  --all-pattern=libbuild2-*
```

To get a list of the standard pre-installed build system modules in a specific version of the toolchain, run:

```
> cd ..\build2-toolchain-X.Y.Z
> dir /B libbuild2-*
```

To verify the build system modules installation you can load them with the following command:

```
> b noop: ..\build2-toolchain-X.Y.Z\tests\libbuild2-*\
```

**5. Clean**

The last thing we need to do is uninstall the staged tools:

```
> cd ..\build2-toolchain-X.Y.Z  # Back to bootstrap.
> b uninstall: build2\ bpkg\
```

## 2.2 Bootstrapping on Windows with Clang

Continuing from Bootstrapping on Windows, there are two common ways to obtain Clang on Windows: bundled with the MSVC installation or as a separate installation. If you are using a separate installation, then the Clang compiler is most likely already in your `PATH` environment variable and, after confirming this is the case, you can continue using the command prompt started on the previous step:

```
> where clang++
```

Otherwise, if you are using Clang that is bundled with MSVC (and haven't manually added its compiler to `PATH`), start the Visual Studio "x64 Native Tools Command Prompt" and set the `PATH` environment variable:

```
> set "PATH=C:\build2\bin;%VCINSTALLDIR%Tools\Llvm\bin;%PATH%"
> where clang++
```

To build with Clang you can either perform the following steps manually or, if after reviewing the steps you are happy with using the defaults, run the `build-clang.bat` batch file. It performs (and echoes) the same set of steps as outlined below but only allows you to customize the installation directory and a few other things (run `build-clang.bat /?` for usage).

For example, you could run this batch file (from the above-mentioned command prompt) like this:

```
> .\build-clang.bat
```

Note that at about half way through (`bpkg fetch` at step 4 below) the script will stop and prompt you to verify the authenticity of the repository certificate. To run the script unattended you can specify the certificate fingerprint with the `--trust` option (see `build-clang.bat /?` for details).

The end result of the bootstrap process (performed either with the script or manually) is the installed toolchain as well as the `bpkg` configuration in `build2-toolchain-X.Y\` that can be used to upgrade to newer versions. It can also be used to uninstall the toolchain:

```
> cd build2-toolchain-X.Y
> bpkg uninstall --all
```

Note that in both cases (manual or scripted bootstrap), if something goes wrong and you need to restart the process, you **must** start with a clean toolchain source by unpacking it afresh from the archive.

The rest of this section outlines the manual bootstrap process.

**1. Bootstrap, Phase 1**

First, we build a minimal build system with the provided `bootstrap-clang.bat` batch file. Normally, the only argument you will pass to this script is the C++ compiler to use but there is also a way to specify compile options; run `bootstrap-clang.bat /?` and see the `build2\INSTALL` file for details.

```
> cd build2
> .\bootstrap-clang.bat clang++ -m64 -w

> build2\b-boot --version
```

Alternatively, we can use the `bootstrap.gmake` GNU makefile to bootstrap in parallel:

```
> cd build2
> mingw32-make -f bootstrap.gmake -j 8 CXX=clang++ "CXXFLAGS=-m64 -w"

> build2\b-boot --version
```

**2. Bootstrap, Phase 2**

Then, we rebuild the build system with the result of Phase 1 linking libraries statically.

```
> build2\b-boot             ^
  "config.cxx=clang++ -m64" ^
  config.bin.lib=static     ^
  build2\exe{b}

> move /y build2\b.exe build2\b-boot.exe

> build2\b-boot --version
```

**3. Stage**

At this step the build system and package manager are built with shared libraries and then staged:

```
> cd ..  # Back to build2-toolchain-X.Y.Z\

> build2\build2\b-boot configure      ^
  "config.cxx=clang++ -m64"           ^
  config.bin.lib=shared               ^
  config.bin.suffix=-stage            ^
  config.install.root=C:\build2       ^
  config.install.data_root=root\stage

> build2\build2\b-boot install: build2\ bpkg\
```

The strange-looking `config.install.data_root=root\stage` means install data files (as opposed to executable files) into the `stage\` subdirectory of wherever `config.install.root` points to (so in our case it will be `C:\build2\stage\`). This subdirectory is temporary and will be removed in a few steps.

Verify that the toolchain binaries can be found and work (this relies on the `PATH` environment variable we have set earlier):

```
> where b-stage
C:\build2\bin\b-stage.exe

> where bpkg-stage
C:\build2\bin\bpkg-stage.exe

> b-stage --version
> bpkg-stage --version
```

At the next step we will use `bpkg` to build and install the entire toolchain. If for some reason you prefer not to build from packages (for example, because the machine is offline), then you can convert this step into a local installation and skip the rest of the steps.

To perform a local installation with the `build-clang.bat` batch file, pass the `--local` option.

To perform a local installation you will need to change the `configure` and `install` command lines above along these lines (see also a note on the following step about only building shared libraries, toolchain executables prefix/suffix, etc):

```
> build2\build2\b-boot configure ^
  config.config.hermetic=true    ^
  "config.cxx=clang++ -m64"      ^
  config.cc.coptions=-O2         ^
  config.bin.lib=shared          ^
  config.install.root=C:\build2

> build2\build2\b-boot install: build2\ bpkg\ bdep\
```

You will also need to build and install the standard build system modules:

```
> b install: !config.install.scope=project libbuild2-*\
```

To verify the build system modules installation you can load them with the following command:

```
> b noop: tests\libbuild2-*\
```

To uninstall such a local installation, run:

```
> b uninstall: build2\ bpkg\ bdep\ libbuild2-*\
```

**4. Install**

Next, we use the staged tools to build and install the entire toolchain from the package repository with the `bpkg` package manager. First, we create the `bpkg` configuration. The configuration values are pretty similar to the previous step and you may want/need to make similar adjustments.

```
> cd ..  # Back to build2-build\
> md build2-toolchain-X.Y
> cd build2-toolchain-X.Y

> bpkg-stage create             ^
  cc                            ^
  config.config.hermetic=true   ^
  "config.cxx=clang++ -m64"     ^
  config.cc.coptions=-O2        ^
  config.bin.lib=shared         ^
  config.install.root=C:\build2
```

The above configuration will only build shared libraries. If you would like to build both shared and static, remove `config.bin.lib=shared`.

To add a custom prefix/suffix to the toolchain executables names, add `config.bin.exe.prefix=...` and/or `config.bin.exe.suffix=...`.

The `config.config.hermetic=true` configuration variable in the above command makes sure the embedded `~host` and `~build2` configurations include the current environment. This is especially important for `~build2` which is used to dynamically build and load ad hoc recipes and build system modules and must therefore match the environment that was used to build the build system itself.

Next, we add the package repository, build, and install:

```
> bpkg-stage add https://pkg.cppget.org/1/alpha
> bpkg-stage fetch
> bpkg-stage build --for install build2 bpkg bdep
> bpkg-stage install --all
```

By default `bpkg` will build the latest available version of each package. You can, however, specify the desired versions explicitly, for example:

```
> bpkg-stage build --for install build2/X.Y.Z bpkg/X.Y.Z bdep/X.Y.Z
```

To verify the result, run:

```
> where b
C:\build2\bin\b.exe

> where bpkg
C:\build2\bin\bpkg.exe

> where bdep
C:\build2\bin\bdep.exe

> b --version
> bpkg --version
> bdep --version
```

Finally, we build and install the standard build system modules:

```
> bpkg build --for install libbuild2-autoconf libbuild2-kconfig
> bpkg install !config.install.scope=project ^
  --all-pattern=libbuild2-*
```

To get a list of the standard pre-installed build system modules in a specific version of the toolchain, run:

```
> cd ..\build2-toolchain-X.Y.Z
> dir /B libbuild2-*
```

To verify the build system modules installation you can load them with the following command:

```
> b noop: ..\build2-toolchain-X.Y.Z\tests\libbuild2-*\
```

**5. Clean**

The last thing we need to do is uninstall the staged tools:

```
> cd ..\build2-toolchain-X.Y.Z  # Back to bootstrap.
> b uninstall: build2\ bpkg\
```

# 2.3 Bootstrapping on Windows with MinGW

Continuing from Bootstrapping on Windows, if you are using your own MinGW distribution, then the resulting `build2` binaries will most likely require a number of DLLs in order to run. It is therefore recommended that you copy the following files from your MinGW `bin\` subdirectory to `C:\build2\bin\` (`*` in the last name will normally be `dw2-1`, `seh-1`, or `sjlj-1`):

```
libwinpthread-1.dll
libstdc++-6.dll
libgcc_s_*.dll
```

To build with MinGW you can either perform the following steps manually or, if after reviewing the steps, you are happy with using the defaults, run the `build-mingw.bat` batch file. It performs (and echoes) the same set of steps as outlined below but only allows you to customize the compiler, installation directory, and a few other things (run `build-mingw.bat /?` for usage).

For example, if your MinGW distribution is in `C:\mingw\`, then you could run it (from the command prompt that we have started earlier) like this:

```
> .\build-mingw.bat C:\mingw\bin\g++
```

If you are using the `build2-mingw` package then you should be able to use just `g++` for the compiler:

```
> .\build-mingw.bat g++
```

If you would like to speed the process up by compiling in parallel, then you can instruct `build-mingw.bat` to bootstrap using GNU make (comes in the `build2-mingw` package), for example:

```
> .\build-mingw.bat --make mingw32-make --make -j8 g++
```

Note that at about half way through (`bpkg fetch` at step 4 below) the script will stop and prompt you to verify the authenticity of the repository certificate. To run the script unattended you can specify the certificate fingerprint with the `--trust` option (see `build-mingw.bat /?` for details).

The end result of the bootstrap process (performed either with the script or manually) is the installed toolchain as well as the `bpkg` configuration in `build2-toolchain-X.Y\` that can be used to upgrade to newer versions. It can also be used to uninstall the toolchain:

```
> cd build2-toolchain-X.Y
> bpkg uninstall --all
```

Note that in both cases (manual or scripted bootstrap), if something goes wrong and you need to restart the process, you **must** start with a clean toolchain source by unpacking it afresh from the archive.

The rest of this section outlines the manual bootstrap process.

**1. Bootstrap, Phase 1**

First, we build a minimal build system with the provided `bootstrap-mingw.bat` batch file. Normally, the only argument you will pass to this script is the C++ compiler to use but there is also a way to specify compile options; run `bootstrap-mingw.bat /?` and see the `build2\INSTALL` file for details.

```
> cd build2
> .\bootstrap-mingw.bat g++ -w -static

> build2\b-boot --version
```

Alternatively, we can use the `bootstrap.gmake` GNU makefile to bootstrap in parallel:

```
> cd build2
> mingw32-make -f bootstrap.gmake -j 8 CXX=g++ CXXFLAGS=-w LDFLAGS=-static

> build2\b-boot --version
```

**2. Bootstrap, Phase 2**

Then, we rebuild the build system with the result of Phase 1 linking libraries statically.

```
> build2\b-boot config.cxx=g++ config.bin.lib=static build2\exe{b}
> move /y build2\b.exe build2\b-boot.exe

> build2\b-boot --version
```

**3. Stage**

At this step the build system and package manager are built with shared libraries and then staged:

```
> cd ..  # Back to build2-toolchain-X.Y.Z\

> build2\build2\b-boot configure      ^
  config.cxx=g++                      ^
  config.bin.lib=shared               ^
  config.bin.suffix=-stage            ^
  config.install.root=C:\build2       ^
  config.install.data_root=root\stage

> build2\build2\b-boot install: build2\ bpkg\
```

The strange-looking `config.install.data_root=root\stage` means install data files (as opposed to executable files) into the `stage\` subdirectory of wherever `config.install.root` points to (so in our case it will be `C:\build2\stage\`). This subdirectory is temporary and will be removed in a few steps.

Verify that the toolchain binaries can be found and work (this relies on the `PATH` environment variable we have set earlier):

```
> where b-stage
C:\build2\bin\b-stage.exe

> where bpkg-stage
C:\build2\bin\bpkg-stage.exe

> b-stage --version
> bpkg-stage --version
```

At the next step we will use `bpkg` to build and install the entire toolchain. If for some reason you prefer not to build from packages (for example, because the machine is offline), then you can convert this step into a local installation and skip the rest of the steps.

To perform a local installation with the `build-mingw.bat` batch file, pass the `--local` option.

To perform a local installation you will need to change the `configure` and `install` command lines above along these lines (see also a note on the following step about only building shared libraries, toolchain executables prefix/suffix, etc):

```
> build2\build2\b-boot configure ^
  config.config.hermetic=true    ^
  config.cxx=g++                 ^
  config.cc.coptions=-O3         ^
  config.bin.lib=shared          ^
  config.install.root=C:\build2

> build2\build2\b-boot install: build2\ bpkg\ bdep\
```

You will also need to build and install the standard build system modules:

```
> b install: !config.install.scope=project libbuild2-*\
```

To verify the build system modules installation you can load them with the following command:

```
> b noop: tests\libbuild2-*\
```

To uninstall such a local installation, run:

```
> b uninstall: build2\ bpkg\ bdep\ libbuild2-*\
```

**4. Install**

Next, we use the staged tools to build and install the entire toolchain from the package repository with the `bpkg` package manager. First, we create the `bpkg` configuration. The configuration values are pretty similar to the previous step and you may want/need to make similar adjustments.

```
> cd ..  # Back to build2-build\
> md build2-toolchain-X.Y
> cd build2-toolchain-X.Y

> bpkg-stage create             ^
  cc                            ^
  config.config.hermetic=true   ^
  config.cxx=g++                ^
  config.cc.coptions=-O3        ^
  config.bin.lib=shared         ^
  config.install.root=C:\build2
```

The above configuration will only build shared libraries. If you would like to build both shared and static, remove `config.bin.lib=shared`.

To add a custom prefix/suffix to the toolchain executables names, add `config.bin.exe.prefix=...` and/or `config.bin.exe.suffix=...`.

The `config.config.hermetic=true` configuration variable in the above command makes sure the embedded `~host` and `~build2` configurations include the current environment. This is especially important for `~build2` which is used to dynamically build and load ad hoc recipes and build system modules and must therefore match the environment

that was used to build the build system itself.

Next, we add the package repository, build, and install:

```
> bpkg-stage add https://pkg.cppget.org/1/alpha
> bpkg-stage fetch
> bpkg-stage build --for install build2 bpkg bdep
> bpkg-stage install --all
```

By default `bpkg` will build the latest available version of each package. You can, however, specify the desired versions explicitly, for example:

```
> bpkg-stage build --for install build2/X.Y.Z bpkg/X.Y.Z bdep/X.Y.Z
```

To verify the result, run (note that the `where` command is not available on Windows XP without the Resource Kit installed):

```
> where b
C:\build2\bin\b.exe

> where bpkg
C:\build2\bin\bpkg.exe

> where bdep
C:\build2\bin\bdep.exe

> b --version
> bpkg --version
> bdep --version
```

Finally, we build and install the standard build system modules:

```
> bpkg build --for install libbuild2-autoconf libbuild2-kconfig
> bpkg install !config.install.scope=project ^
  --all-pattern=libbuild2-*
```

To get a list of the standard pre-installed build system modules in a specific version of the toolchain, run:

```
> cd ..\build2-toolchain-X.Y.Z
> dir /B libbuild2-*
```

To verify the build system modules installation you can load them with the following command:

```
> b noop: ..\build2-toolchain-X.Y.Z\tests\libbuild2-*\
```

**5. Clean**

The last thing we need to do is uninstall the staged tools:

```
> cd ..\build2-toolchain-X.Y.Z  # Back to bootstrap.
> b uninstall: build2\ bpkg\
```

# 3 Bootstrapping on Mac OS X

The `build2` toolchain requires Mac OS version 10.5 (Leopard) or later. We will also be using the system C++ toolchain that comes with the Xcode Command Line Tools. You should be able to use other/custom C++ toolchains, however, this is the only configuration that is tested and guaranteed to work.

To verify that Command Line Tools are installed, run:

```
$ clang++ --version
```

It should produce something along these lines:

```
Apple LLVM version X.Y.Z (clang-A.B.C) (based on LLVM M.N.P)
```

To install Command Line Tools, run:

```
$ xcode-select --install
```

Once this is done continue with Bootstrapping on UNIX.

# 4 Bootstrapping on UNIX

The following instructions are for bootstrapping `build2` on UNIX-like operating systems (GNU/Linux, FreeBSD, etc). For Mac OS X first see Bootstrapping on Mac OS X. These instructions should also be used for UNIX emulation layers on Windows (for example, WSL, MSYS, or Cygwin) where you already have a UNIX shell with standard utilities.

**1. Create Build Directory**

You will want to keep this directory around in order to upgrade to new toolchain versions in the future. In this guide we use `~/build2-build/` as the build directory and `/usr/local/` as the installation directory but you can use other paths.

```
$ cd
$ mkdir build2-build
$ cd build2-build
```

**2. Download, Verify, and Unpack**

Download `build2-toolchain-X.Y.Z.tar.xz` (or its `.tar.gz` variant if you don't have **`xz(1)`**) as well as its `.sha256` checksum from Download page.

Place everything into `~/build2-build/` (build directory) and verify the archive checksum matches:

```
# Linux, WSL, MSYS, Cygwin:
#
$ sha256sum -c build2-toolchain-X.Y.Z.tar.xz.sha256

# Mac OS X:
#
$ shasum -a 256 -c build2-toolchain-X.Y.Z.tar.xz.sha256

# FreeBSD (compare visually):
#
$ cat build2-toolchain-X.Y.Z.tar.xz.sha256
$ sha256 -r build2-toolchain-X.Y.Z.tar.xz
```

Unpack the archive and change to its directory:

```
> tar -xf build2-toolchain-X.Y.Z.tar.xz
> cd build2-toolchain-X.Y.Z
```

Next you can either perform the rest of the steps manually or, if after reviewing the steps, you are happy with using the defaults, run the `build.sh` shell script. It performs (and echoes) the same set of steps as outlined below but only allows you to customize the compiler, installation directory, and a few other things (run `build.sh -h` for usage).

For example, this command will use `g++` and install the toolchain into `/usr/local/`.

```
$ ./build.sh g++
```

While this will use Clang and install into `/opt/build2`:

```
$ ./build.sh --install-dir /opt/build2 --sudo sudo clang++
```

If you would like to speed the process up by compiling in parallel, then you can instruct `build.sh` to bootstrap using GNU make (can be called `gmake` instead of `make` on some platforms), for example:

```
$ ./build.sh --make make --make -j8 g++
```

Note that at about half way through (`bpkg fetch` at step 4 below) the script will stop and prompt you to verify the authenticity of the repository certificate. To run the script unattended you can specify the certificate fingerprint with the `--trust` option (see `build.sh -h` for details).

The end result of the bootstrap process (performed either with the script or manually) is the installed toolchain as well as the `bpkg` configuration in `build2-toolchain-X.Y/` that can be used to upgrade to newer versions. It can also be used to uninstall the toolchain:

```
$ cd build2-toolchain-X.Y
$ bpkg uninstall --all
```

Note that in both cases (manual or scripted bootstrap), if something goes wrong and you need to restart the process, you **must** start with a clean toolchain source by unpacking it afresh from the archive.

The rest of this section outlines the manual bootstrap process.

**1. Bootstrap, Phase 1**

First, we build a minimal build system with the provided `bootstrap.sh` script. Normally, the only argument you will pass to this script is the C++ compiler to use but there is also a way to specify compile options and a few other things; run `bootstrap.sh -h` and see the `build2/INSTALL` file for details.

```
$ cd build2
$ ./bootstrap.sh g++ -w

$ build2/b-boot --version
```

Alternatively, we can use the `bootstrap.gmake` GNU makefile to bootstrap in parallel:

```
$ cd build2
$ make -f bootstrap.gmake -j 8 CXX=g++ CXXFLAGS=-w

$ build2/b-boot --version
```

**2. Bootstrap, Phase 2**

Then, we rebuild the build system with the result of Phase 1 linking libraries statically.

```
$ build2/b-boot config.cxx=g++ config.bin.lib=static build2/exe{b}
$ mv build2/b build2/b-boot

$ build2/b-boot --version
```

**3. Stage**

At this step the build system and package manager are built with shared libraries and then staged. Here you may want to adjust a few things, such as the installation directory or the `sudo` program (remove the `config.install.sudo` line if you don't need one).

You may also need to remove the `config.bin.rpath` line if your target doesn't support *rpath*. Specifically, if building on Windows (with MSYS or Cygwin), remove both `.rpath` and `.sudo`. But if unsure, leave `.rpath` in – if your target doesn't support it, you will get an error and will need to reconfigure without it.

```
$ cd ..  # Back to build2-toolchain-X.Y.Z/

$ build2/build2/b-boot configure      \
  config.cxx=g++                      \
  config.bin.lib=shared               \
```

```
  config.bin.suffix=-stage            \
  config.bin.rpath=/usr/local/lib     \
  config.install.root=/usr/local      \
  config.install.data_root=root/stage \
  config.install.sudo=sudo

$ build2/build2/b-boot install: build2/ bpkg/
```

The above command will build all the dependencies of `build2` and `bpkg` from sources bundled with `build2-toolchain`. If instead you would like to use system-installed versions for some of them, then you can specify empty `config.import.*` values to disable the use of the bundled versions. For example, to use the system-installed SQLite:

```
$ build2/build2/b-boot configure \
  ...                            \
  config.import.libsqlite3=
```

If performing an installation with the `build.sh` script, then to use the system-installed dependencies pass the `--system` option, specifying such dependencies as a comma-separated list. For example:

```
$ ./build.sh --system libsqlite3,libpkg-config g++
```

The strange-looking `config.install.data_root=root/stage` means install data files (as opposed to executable files) into the `stage/` subdirectory of wherever `config.install.root` points to (so in our case it will be `/usr/local/stage/`). Note that this subdirectory is temporary and will be removed in a few steps. But if you don't like the default location, feel free to change it (for example, to `/tmp/stage`).

Depending on the installation directory, the installed `build2` binaries may not be automatically found. On most platforms `/usr/local/bin/` is in the `PATH` environment variable by default and you should be able to run:

```
$ which b-stage
/usr/local/bin/b-stage

$ which bpkg-stage
/usr/local/bin/bpkg-stage

$ b-stage --version
$ bpkg-stage --version
```

If, however, you installed, say, into `/opt/build2`, then you will need to add its `bin/` subdirectory to `PATH` (re-run the above commands to verify):

```
$ export PATH="/opt/build2/bin:$PATH"
```

Strictly speaking this is not absolutely necessary and you can adjust the rest of the commands to use absolute paths. This, however, does not make for very readable examples so below we assume the installation's `bin/` subdirectory is in `PATH`.

At the next step we will use `bpkg` to build and install the entire toolchain. If for some reason you prefer not to build from packages (for example, because the machine is offline), then you can convert this step into a local installation and skip the rest of the steps.

To perform a local installation with the `build.sh` script, pass the `--local` option.

To perform a local installation you will need to change the `configure` and `install` command lines above along these lines (see also notes on the following step about only building shared libraries, private installation subdirectory, toolchain executables prefix/suffix, etc):

```
$ build2/build2/b-boot configure         \
  config.config.hermetic=true            \
  config.cxx=g++                         \
  config.cc.coptions=-O3                 \
  config.bin.lib=shared                  \
  config.bin.rpath=/usr/local/lib/build2 \
  config.install.root=/usr/local         \
  config.install.private=build2          \
  config.install.sudo=sudo

$ build2/build2/b-boot install: build2/ bpkg/ bdep/
```

You will also need to build and install the standard build system modules:

```
$ b install: '!config.install.scope=project' libbuild2-*/
```

To verify the build system modules installation you can load them with the following command:

```
$ b noop: tests/libbuild2-*/
```

To uninstall such a local installation, run:

```
$ b uninstall: build2/ bpkg/ bdep/ libbuild2-*/
```

**4. Install**

Next, we use the staged tools to build and install the entire toolchain from the package repository with the `bpkg` package manager. First, we create the `bpkg` configuration. The configuration values are pretty similar to the previous step and you may want/need to make similar adjustments.

```
$ cd ..  # Back to build2-build/
$ mkdir build2-toolchain-X.Y
$ cd build2-toolchain-X.Y

$ bpkg-stage create                        \
  cc                                       \
  config.config.hermetic=true              \
  config.cxx=g++                           \
  config.cc.coptions=-O3                   \
  config.bin.lib=shared                    \
  config.bin.rpath=/usr/local/lib/build2 \
  config.install.root=/usr/local           \
  config.install.private=build2            \
  config.install.sudo=sudo
```

The above configuration will only build shared libraries. If you would like to build both shared and static, remove `config.bin.lib=shared`.

The above configuration will install shared libraries that `build2` depends on into a private subdirectory. This is primarily useful when installing into a shared location, such as `/usr/local/`. By hiding the libraries in the private subdirectory we make sure that they will not interfere with anything that is already installed into such a shared location and that any further such installations won't interfere with `build2`. If, however, you are installing into a private location, such as `/opt/build2/`, then you can remove `config.install.private=build2`.

To add a custom prefix/suffix to the toolchain executables names, add `config.bin.exe.prefix=...` and/or `config.bin.exe.suffix=...`.

The `config.config.hermetic=true` configuration variable in the above command makes sure the embedded `~host` and `~build2` configurations include the current environment. This is especially important for `~build2` which is used to dynamically build and load ad hoc recipes and build system modules and must therefore match the environment that was used to build the build system itself.

Next, we add the package repository, build, and install:

```
$ bpkg-stage add https://pkg.cppget.org/1/alpha
$ bpkg-stage fetch
$ bpkg-stage build --for install build2 bpkg bdep
$ bpkg-stage install --all
```

The above command will build all the dependencies of `build2`, `bpkg`, and `bdep` from source packages. If instead you would like to use system-installed versions for some of them, then you can list them with the `sys` scheme to make `bpkg-stage` treat them as available from the system rather than building them from source. For example, to use the system-installed SQLite:

```
$ bpkg-stage build --for install build2 bpkg bdep ?sys:libsqlite3
```

By default `bpkg` will build the latest available version of each package. You can, however, specify the desired versions explicitly, for example:

```
$ bpkg-stage build --for install build2/X.Y.Z bpkg/X.Y.Z bdep/X.Y.Z
```

To verify the result, run:

```
$ which b
/usr/local/bin/b

$ which bpkg
/usr/local/bin/bpkg

$ which bdep
/usr/local/bin/bdep

$ b --version
$ bpkg --version
$ bdep --version
```

Finally, we build and install the standard build system modules:

```
$ bpkg build --for install libbuild2-autoconf libbuild2-kconfig
$ bpkg install '!config.install.scope=project' \
  --all-pattern=libbuild2-*
```

To get a list of the standard pre-installed build system modules in a specific version of the toolchain, run:

```
$ cd ../build2-toolchain-X.Y.Z
$ ls -d libbuild2-*
```

To verify the build system modules installation you can load them with the following command:

```
$ b noop: ../build2-toolchain-X.Y.Z/tests/libbuild2-*/
```

**5. Clean**

The last thing we need to do is uninstall the staged tools:

```
$ cd ../build2-toolchain-X.Y.Z  # Back to bootstrap.
$ b uninstall: build2/ bpkg/
```

# 5 Upgrading

At this point we assume that you have the build2 toolchain installed and would like to upgrade it to a newer version. We also expect that you have the toolchain `bpkg` configuration in the `build2-toolchain-X.Y/` directory, as produced by the bootstrap process. If you don't have the `bpkg` configuration but do have the toolchain installed somehow (for example, using your distribution's package manager), then you can create the configuration as shown at the end. If you have neither, then you will need to go through the bootstrap process.

There are two ways to upgrade: dirty (but quick) and staged (but more involved). In the *dirty upgrade* we override the existing installation without first uninstalling it. If some installed files no longer exist in the new version, they will remain "installed" until cleaned up manually. Also, with this approach we never get a chance to make sure the new build is functional.

In the *staged upgrade* we first install a `-stage` build of the new toolchain (similar to what we did during bootstrap), test it, uninstall the old toolchain, install the new toolchain as "final", and finally uninstall `-stage`.

We recommend that you use a dirty upgrade for toolchain patch releases with the same `X.Y` (`MAJOR.MINOR`) version and a staged upgrade otherwise. With patch releases we guarantee not to alter the installation file set.

Without periodic upgrades your version of the toolchain may become too old to be able to upgrade itself. In this case you will have to fall back onto the bootstrap process.

The below upgrade process does not cover upgrading the `baseutils` and `mingw` packages on Windows (see Bootstrapping on Windows for details). We recommend using the bootstrap process to upgrade these packages since all the straightforward upgrade sequences would lead to either the old toolchain using the new utilities or vice versa.

For both ways of upgrading we need to make sure that the build system modules are built and installed with the new version of the toolchain. The set of build system modules can also change from version to version.

If using the Windows command prompt, the `!config.install.scope=project` command line argument should not be quoted.

The dirty upgrade is straightforward:

```
$ cd build2-toolchain-X.Y
$ bpkg uninstall '!config.install.scope=project' \
  --all-pattern=libbuild2-*
$ bpkg drop --all-pattern=libbuild2-*
$ bpkg fetch
$ bpkg build --for install -pr
$ bpkg install --all
$ bpkg build --for install libbuild2-autoconf libbuild2-kconfig
$ bpkg install '!config.install.scope=project' \
  --all-pattern=libbuild2-*
```

The `-pr` options stands for `--patch` and `--recursive` – upgrade the built packages and their dependencies to the latest patch version, recursively. See **bpkg-pkg-build(1)** for details.

You can also issue the `status` command after `fetch` to examine which versions are available. The above `build` command will upgrade all the packages to the latest available patch versions but you can override this by specifying the desired packages and/or versions explicitly, for example:

```
$ bpkg status
!build2 configured 1.0.0 available 1.0.1 1.0.2 2.0.0
...

$ bpkg build --for install build2/1.0.1
```

The staged upgrade consists of several steps:

**0. Check for Updates**

There is no harm in running `bpkg fetch` in the existing configuration so we can use it to determine if any updates are available, whether we can use the simpler dirty upgrade, and, if not, the target `X.Y` (`MAJOR.MINOR`) version for the staged upgrade:

```
$ cd build2-toolchain-X.Y
$ bpkg fetch
$ bpkg status build2 bpkg bdep
```

Let's say the new version is X.Z.

**1. Create New Configuration**

First we make a copy of the old configuration. We will need the original later to cleanly uninstall the old toolchain, and, maybe, to rollback the installation if the new version doesn't work out.

```
$ cd ..
$ cp -rp build2-toolchain-X.Y build2-toolchain-X.Z
```

Or, using Windows command prompt:

```
> cd ..
> xcopy /s /q /i build2-toolchain-X.Y build2-toolchain-X.Z
```

**2. Build and Install as `-stage`**

This step is similar to the dirty upgrade except that we use the copied configuration, upgrade (`--upgrade|-u`) instead of patching (`--patch|-p`), and install the new toolchain with the `-stage` suffix:

```
$ cd build2-toolchain-X.Z
$ bpkg drop --all-pattern=libbuild2-*
$ bpkg build --for install -ur
```

Once this is done, we can proceed to installing:

```
$ bpkg install                          \
  config.bin.suffix=-stage              \
  config.install.data_root=root/stage \
  --all
```

If during installation you have added a custom prefix/suffix to the toolchain executables names with `config.bin.exe.prefix` and/or `config.bin.exe.suffix`, add `config.bin.exe.prefix=[null]` and/or `config.bin.exe.suffix=[null]` to suppress them in the executables being staged.

You can also specify the desired packages and/or versions explicitly, again, similar to the dirty upgrade.

**3. Test Staged**

Now you can test the new toolchain on your projects, etc. Remember to use the `-stage`-suffixed binaries (`bdep-stage` will automatically use `bpkg-stage` which in turn will use `b-stage`):

```
$ b-stage --version
$ bpkg-stage --version
$ bdep-stage --version
```

**4. Uninstall Old, Install New**

Once we are satisfied that the new toolchain works, we can uninstall the old one and install the new one:

```
$ cd ../build2-toolchain-X.Y
$ bpkg uninstall --all

$ cd ../build2-toolchain-X.Z
$ bpkg-stage install --all
$ bpkg build --for install libbuild2-autoconf libbuild2-kconfig
$ bpkg install '!config.install.scope=project' \
  --all-pattern=libbuild2-*
```

**5. Uninstall Staged**

Finally, we clean up by removing the staged toolchain (hint: use the command line history to find the corresponding `install` command and change it to `uninstall`; see also a note at step 2 about toolchain executables prefix/suffix):

```
$ bpkg uninstall                      \
  config.bin.suffix=-stage            \
  config.install.data_root=root/stage \
  --all
```

You can also remove the old configuration in `build2-toolchain-X.Y/` if you think you no longer need it.

If you ever need to (re-)create the `bpkg` configuration for the toolchain from scratch, it is fairly simple (you may need to adjust the compiler, options, installation directory, etc; see the bootstrap steps for details):

For UNIX-like operating systems (GNU/Linux, Mac OS X, FreeBSD, etc):

```
$ bpkg-stage create             \
cc                              \
config.config.hermetic=true     \
config.cxx=g++                  \
config.cc.coptions=-O3          \
config.bin.lib=shared           \
config.bin.rpath=/usr/local/lib \
config.install.root=/usr/local  \
config.install.private=build2   \
config.install.sudo=sudo
```

For Windows with MSVC (from the Visual Studio "x64 Native Tools Command Prompt"):

```
> bpkg-stage create             ^
  cc                            ^
  config.config.hermetic=true   ^
  config.cxx=cl                 ^
  config.cc.coptions=/O2        ^
  config.bin.lib=shared         ^
  config.install.root=C:\build2
```

For Windows with Clang (from a suitable command prompt, see Bootstrapping on Windows with Clang for details):

```
> bpkg-stage create             ^
  cc                            ^
  config.config.hermetic=true   ^
  "config.cxx=clang++ -m64"     ^
  config.cc.coptions=-O2        ^
  config.bin.lib=shared         ^
  config.install.root=C:\build2
```

For Windows with MinGW (from the command prompt):

```
> bpkg-stage create             ^
  cc                            ^
  config.config.hermetic=true   ^
  config.cxx=g++                ^
  config.cc.coptions=-O3        ^
  config.bin.lib=shared         ^
  config.install.root=C:\build2
```