The `build2` Package Manager

Revision `0.15`, July 2022
This revision of the document describes the `build2` package manager `0.15.x` series.

Table of Contents

Preface

This document describes `bpkg`, the `build2` package dependency manager. For the package manager command line interface refer to the **`bpkg(1)`** man pages.

1 Package Name

The `bpkg` package name can contain ASCII alphabetic characters (`[a-zA-Z]`), digits (`[0-9]`), underscores (_), plus/minus (+-), and dots/periods (`.`). The name must be at least two characters long with the following additional restrictions:

1. It must start with an alphabetic character.
2. It must end with an alphabetic, digit, or plus character.
3. It must not be any of the following illegal names:

```
build
con prn aux nul
com1 com2 com3 com4 com5 com6 com7 com8 com9
lpt1 lpt2 lpt3 lpt4 lpt5 lpt6 lpt7 lpt8 lpt9
```

The use of the plus (+) character in package names is discouraged. Pluses are used in URL encoding which makes specifying packages that contain pluses in URLs cumbersome.

The use of the dot (`.`) character in package names is discouraged except for distinguishing the implementations of the same functionality for different languages. For example, `libfoo` and `libfoo.bash`.

Package name comparison is case-insensitive but the original case must be preserved for display, in file names, etc. The reason for case-insensitive comparison is Windows file names.

If the package is a library then it is strongly recommended that you start its package name with the `lib` prefix, for example, `libfoo`. Some package repositories may make this a requirement as part of their submission policy.

If a package (normally a library) supports usage of multiple major versions in the same project, then it is recommended to append the major version number to the package name starting from version `2.0.0`, for example, `libfoo` (before `2.0.0`), `libfoo2` (`2.Y.Z`), `libfoo3` (`3.Y.Z`), etc.

2 Package Version

The `bpkg` package version format tries to balance the need of accommodating existing software versions on one hand and providing a reasonably straightforward comparison semantics on another. For some background on this problem see **`deb-version(1)`** and the Semantic Versioning specification.

Note also that if you are starting a new project that will use the `build2` toolchain, then it is strongly recommended that you use the *standard versioning* scheme which is a more strictly defined subset of semantic versioning that allows automation of many version management tasks. See `version` Module for details.

The `bpkg` package version has the following form:

```
[+<epoch>-]<upstream>[-<prerel>][+<revision>][#<iteration>]
```

The *epoch* part should be an integer. It can be used to change to a new versioning scheme that would be incompatible with the old one. If not specified, then *epoch* defaults to `1` except for a stub version (see below) in which case it defaults to `0`. The explicit zero *epoch* can be used if the current versioning scheme (for example, date-based) is known to be temporary.

The *upstream* part is the upstream software version that this package is based on. It can only contain alpha-numeric characters and `.`. The `.` character is used to separate the version into *components*.

The *prerel* part is the upstream software pre-release marker, for example, alpha, beta, candidate, etc. Its format is the same as for *upstream* except for two special values: the absent *prerel* (for example, `1.2.3`) signifies the maximum or final release while the empty *prerel* (for example, `1.2.3-`) signifies the minimum or earliest possible release. The minimum release is intended to be used for version constraints (for example, `libfoo < 1.2.3-`) rather than actual releases.

The *revision* part should be an integer. It is used to version package releases that are based on the same upstream versions. If not specified, then *revision* defaults to `0`.

The *iteration* part is an integer. It is used internally by `bpkg` to automatically version modifications to the packaging information (specifically, to package manifest and lockfile) in *external packages* that have the same upstream version and revision. As a result, the *iteration* cannot not be specified by the user and is only shown in the `bpkg` output (for example, by `pkg-status` command) in order to distinguish between package iterations with otherwise identical versions. Note also that *iteration* is relative to the `bpkg` configuration. Or, in other words, it is an iteration number of a package as observed by a specific configuration. As a result, two configurations can "see" the same package state as two different iterations.

Package iterations are used to support package development during which requiring the developer to manually increment the version or revision after each modification would be impractical. This mechanism is similar to the automatic commit versioning provided by the *standard version* except that it is limited to the packaging information but works for uncommitted changes.

Version `+0-0-` (least possible version) is reserved and specifying it explicitly is illegal. Explicitly specifying this version does not make much sense since `libfoo < +0-0-` is always false and `libfoo > +0-0-` is always true. In the implementation this value is used as a special empty version.

Version `0` (with a potential revision, for example, `0+1`, `0+2`) is used to signify a *stub package*. A stub is a package that does not contain source code and can only be "obtained" from other sources, for example, a system package manager. Note that at some point a stub may be converted into a full-fledged package at which point it will be assigned a "real" version. It is assumed that this version will always be greater than the stub version.

When displaying the package version or when using the version to derive the file name, the default *epoch* value as well as zero *revision* and *iteration* values are omitted (even if they were explicitly specified, for instance, in the package manifest). For example, `+1-1.2.3+0` will be used as `libfoo-1.2.3`.

This versioning scheme and the choice of delimiter characters (`.-+`) is meant to align with semantic versioning.

Some examples of versions:

```
0+1
+0-20180112
1.2.3
1.2.3-a1
1.2.3-b2
1.2.3-rc1
1.2.3-alpha1
1.2.3-alpha.1
1.2.3-beta.1
1.2.3+1
+2-1.2.3
+2-1.2.3-alpha.1+3
+2.2.3#1
1.2.3+1#1
+2-1.2.3+1#2
```

The version sorting order is *epoch*, *upstream*, *prerel*, *revision*, and finally, *iteration*. The *upstream* and *prerel* parts are compared from left to right, one component at a time, as described next.

To compare two components, first the component types are determined. A component that only consists of digits is an integer. Otherwise, it is a string. If both components are integers, then they are compared as integers. Otherwise, they are compared lexicographically and case-insensitively. The reason for case-insensitive comparison is Windows file names.

A non-existent component is considered 0 if the other component is an integer and an empty string if the other component is a string. For example, in `1.2` vs `1.2.0`, the third component in the first version is 0 and the two versions are therefore equal. As a special exception to this rule, an absent *prerel* part is always greater than any non-absent part. And thus making the final release always older than any pre-release.

This algorithm gives correct results for most commonly-used versioning schemes, for example:

```
1.2.3 < 12.2
1.alpha < 1.beta
20151128 < 20151228
2015.11.28 < 2015.12.28
```

One notable versioning scheme where this approach gives an incorrect result is hex numbers (consider `A` vs `1A`). The simplest work around is to convert such numbers to decimal. Alternatively, one can fix the width of the hex number and pad all the values with leading zeros, for example: `00A` vs `01A`.

It is also possible to convert the *upstream* and *prerel* parts into a *canonical representation* that will produce the correct comparison result when always compared lexicographically and as a whole. This can be useful, for example, when storing versions in the database which would otherwise require a custom collation implementation to obtain the correct sort order.

To convert one of these parts to its canonical representation, all its string components are converted to the lower case while all its integer components are padded with leading zeros to the fixed length of `16` characters, with all trailing zero-only components removed. Note that this places an implementation limit on the length of integer components which should be checked by the implementation when converting to the canonical representation. The `16` characters limit was chosen to still be able to represent (with some spare) components in the *YYYYMMDDhhmmss* form while not (visually) bloating the database too much. As a special case, the absent *prerel* part is represented as ~. Since the ASCII code for ~ is greater than any other character that could appear in *prerel*, such a string will always be greater than any other representation. The empty *prerel* part is represented as an empty string.

Note that because it is not possible to perform a reverse conversion without the possibility of loss (consider `01.AA.BB`), the original parts may also have to be stored, for example, for display, to derive package archive names, etc.

In quite a few contexts the implementation needs to ignore the *revision* and/or *iteration* parts. For example, this is needed to implement the semantics of newer revisions/iterations of packages replacing their old ones since we do not keep multiple revisions/iterations of the same upstream version in the same repository. As a result, in the package object model, we have a version key as just {*epoch*, *upstream*, *prerel*} but also store the package revision and iteration so that it can be shown to the user, etc.

3 Package Version Constraint

The `bpkg` package version constraint may follow the package name in certain contexts, such as the manifest values and `bpkg` command line, to restrict the allowed package version set. It can be specified using comparison operators, shortcut (to range) operators, or ranges and has the following form:

```
<version-constraint> = <comparison> | <shortcut> | <range>
<comparison>         = ('==' | '>' | '<' | '>=' | '<=') <version>
<shortcut>           = ('^' | '~') <version>
<range>              = ('(' | '[') <version> <version> (')' | ']')
```

The shortcut operators can only be used with standard versions (a semantic version without the pre-release part is a standard version). They are equivalent to the following ranges. The `X.Y.Z-` version signifies the earliest pre-release in the `X.Y.Z` series; see Package Version for details.

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

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

That is, the tilde (~) constraint allows upgrades to any further patch version while the caret (^) constraint – also to any further minor version.

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

Note that the shortuct operators can only be used with the complete, three-component versions (`X.Y.Z` with the optional pre-release part per the standard version). Specifically, there is no support for special `^X.Y` or `~X` semantics offered by some package manager – if desired, such functionality can be easily achieved with ranges. Also, the `0.0.Z` version is not considered special except as having zero major component for the tilde semantics discussed above.

Note also that pre-releases do not require any special considerations when used with the shortcut operators. For example, if package `libfoo` is usable starting with the second beta of the `2.0.0` release, then our constraint could be expressed as:

```
libfoo ^2.0.0-b.2
```

Internally, shortcuts and comparisons can be represented as ranges (that is, `[v, v]` for ==, `(v, inf)` for >, etc). However, for display and serialization such representations should be converted back to simple operators. While it is possible that the original manifest specified equality or shortucts as full ranges, it is acceptable to display/serialize them as simpler operators.

Instead of a concrete value, the version in the constraint can be specified in terms of the dependent package's version (that is, the version of the package placing the constraint) using the special `$` value. For example:

```
libfoo == $
```

A constraint that contains `$` is called incomplete. This mechanism is primarily useful when developing related packages that should track each other's versions exactly or closely.

In comparison operators and ranges the `$` value is replaced with the dependent version ignoring the revision. For shortcut operators, the dependent version must be a standard version and the following additional processing is applied depending on whether the version is a release, final pre-release, or a snapshot pre-release.

1. For a release we set the min version patch to zero. For `^` we also set the minor version to zero, unless the major version is zero (reduces to `~`). The max version is set according to the standard shortcut logic. For example, `~$` is completed as follows:

   ```
   1.2.0 -> [1.2.0 1.3.0-)
   1.2.1 -> [1.2.0 1.3.0-)
   1.2.2 -> [1.2.0 1.3.0-)
   ```

   And `^$` is completed as follows:

   ```
   1.0.0 -> [1.0.0 2.0.0-)
   1.1.1 -> [1.0.0 2.0.0-)
   ```

2. For a final pre-release the key observation is that if the patch component for `~` or minor and patch components for `^` are not zero, then that means there has been a compatible release and we treat this case the same as release, ignoring the pre-release part. If, however, it/they are zero, then that means there may yet be no final release and we have to start from the first alpha. For example, for the `~$` case:

   ```
   1.2.0-a.1 -> [1.2.0-a.1 1.3.0-)
   1.2.0-b.2 -> [1.2.0-a.1 1.3.0-)
   1.2.1-a.1 -> [1.2.0     1.3.0-)
   1.2.2-b.2 -> [1.2.0     1.3.0-)
   ```

   And for the `^$` case:

   ```
   1.0.0-a.1 -> [1.0.0-a.1 2.0.0-)
   1.0.0-b.2 -> [1.0.0-a.1 2.0.0-)
   1.0.1-a.1 -> [1.0.0     2.0.0-)
   1.1.0-b.2 -> [1.0.0     2.0.0-)
   ```

3. For a snapshot pre-release we distinguish two cases: a patch snapshot (the patch component is not zero) and a major/minor snapshot (the patch component is zero). For the patch snapshot case we assume that it is (most likely) developed independently of the dependency and we treat it the same as the final pre-release case. For example, if the dependent version is `1.2.1-a.0.nnn`, the dependency could be `1.2.0` or `1.2.2` (or somewhere in-between).

   For the major/minor snapshot we assume that all the packages are developed in the lockstep and have the same `X.Y.0` version. In this case we make the range start from the earliest possible version in this "snapshot series" and end before the final pre-release. For example (in this case `~` and `^` are treated the same):

   ```
   1.2.0-a.0.nnn -> [1.2.0-a.0.1 1.2.0-a.1)
   2.0.0-b.2.nnn -> [2.0.0-b.2.1 2.0.0-b.3)
   ```

4 Package Build System Skeleton

There are situations where `bpkg` may need to evaluate `buildfile` expressions and fragments before committing to a particular version of the package and therefore before actually unpacking anything. For example, `bpkg` may need to evaluate a condition in the conditional dependency or it may need to negotiate a configuration among several dependents of a

package which requires it to know this package's configuration variable types and default values.

To solve this chicken and egg kind of problem, `bpkg` includes a minimal subset of the build system files along with the package's standard metadata (name, version, etc) into the repository metadata (`packages.manifest`). This subset is called the package build system skeleton, or just package skeleton for short, and includes the `build/bootstrap.build` and `build/root.build` files (or their alternative naming scheme variants) as well as any files that may be sourced by `root.build`.

The inclusion of `build/bootstrap.build` and `build/root.build` (if present) as well as any `build/config/*.build` (or their alternative naming scheme variants) is automatic. However, if `root.build` sources any files other than `build/config/*.build`, then they must be specified explicitly in the package manifest using the `build-file` value.

Inside these buildfiles the skeleton load can be distinguished from normal load by examining the `build.mode` variable, which is set to `skeleton` during the skeleton load. In particular, this variable must be used to omit loading of build system modules that are neither built-in nor standard pre-installed and which are therefore listed as package dependencies. Such modules are not yet available during the skeleton load. For example:

```
# root.build

using cxx       # Ok, built-in module.
using autoconf  # Ok, standard pre-installed module.

if ($build.mode != 'skeleton')
  using hello
```

The `build.mode` variable can also be used to omit parts of `root.build` that are expensive to evaluate and which are only necessary during the actual build. Here is a realistic example:

```
# root.build

...

using cxx

# Determine the GCC plugin directory. But omit doing it during the
# skeleton load.
#
if ($build.mode != 'skeleton')
{
  if ($cxx.id != 'gcc')
    fail 'this project can only be built with GCC'

  # If plugin support is disabled, then -print-file-name will print
  # the name we have passed (the real plugin directory will always
  # be absolute).
  #
  plugin_dir = [dir_path] \
    $process.run($cxx.path -print-file-name=plugin)
```

```
  if ("$plugin_dir" == plugin)
    fail "$recall($cxx.path) does not support plugins"

  plugin_dir = $normalize($plugin_dir)
}
```

5 Dependency Configuration Negotiation

In `bpkg`, a dependent package may specify a desired configuration for a dependency package. Because there could be multiple such dependents, `bpkg` needs to come up with a dependency configuration that is acceptable to all of them. This process is called the dependency configuration negotiation.

The desired dependency configuration is specified as part of the `depends` manifest value and can be expressed as either a single `require` clause or as a pair of `prefer`/`accept` clauses.

The `require` clause is essentially a shortcut for specifying the `prefer`/`accept` clauses where the `accept` condition simply verifies all the variable values assigned in the `prefer` clause. It is, however, further restricted to the common case of only setting `bool` variables and only to `true` to allow additional optimizations during the configuration negotiation. The remainder of this section only deals with the general `prefer`/`accept` semantics.

While the exact format of `prefer`/`accept` is described as part of the `depends` manifest value, for this section it is sufficient to know that the `prefer` clause is an arbitrary `buildfile` fragment that is expected to set one or more dependency configuration variables to the values preferred by this dependent while the `accept` clause is a `buildfile` eval context expression that should evaluate to `true` or `false` indicating whether the dependency configuration values it is evaluated on are acceptable to this dependent. For example:

```
libfoo ^1.0.0
{
  # We prefer the cache but can work without it.
  # We need the buffer of at least 4KB.
  #
  prefer
  {
    config.libfoo.cache = true

    config.libfoo.buffer = ($config.libfoo.buffer < 4096 \
                            ? 4096                       \
                            : $config.libfoo.buffer)
  }

  accept ($config.libfoo.buffer >= 4096)
}
```

The configuration negotiation algorithm can be summarized as cooperative refinement. Specifically, whenever a `prefer` clause of a dependent changes any configuration value, all other dependents' `prefer` clauses are re-evaluated. This process continues until there are no more changes (success), one of the `accept` clauses returned `false` (failure), or the process starts "yo-yo'ing" between two or more configurations (failure).

The dependents are expected to cooperate by not overriding "better" values that were set by other dependents. Consider the following two `prefer` clauses:

```
prefer
{
  config.libfoo.buffer = 4096
}

prefer
{
  config.libfoo.buffer = ($config.libfoo.buffer < 4096 \
                          ? 4096                       \
                          : $config.libfoo.buffer)
}
```

The first version is non-cooperative and should only be used if this dependent requires the buffer to be exactly 4KB. The second version is cooperative: it will increase the buffer to the minimum required by this dependent but will respect values above 4KB.

One case where we don't need to worry about this is when setting the configuration variable to the "best" possible value. One common example of this is setting a `bool` configuration to `true`.

With a few exceptions discussed below, a dependent must always re-set the configuration variable, even if to the better value. For example, the following is an incorrect attempt at the above cooperative `prefer` clause:

```
prefer
{
  if ($config.libfoo.buffer < 4096)  # Incorrect.
    config.libfoo.buffer = 4096
}
```

The problem with the above attempt is that the default value could be greater than 4KB, in which case `bpkg` will have no idea that there is a dependent relying on this configuration value.

Before each `prefer` clause re-evaluation, variables that were first set to their current values by this dependent are reset to their defaults thus allowing the dependent to change its mind, for instance, in response to other configuration changes. For example:

```
# While we have no preference about the cache, if enabled/disabled,
# we need a bigger/smaller buffer.
#
prefer
{
  min_buffer = ($config.libfoo.cache ? 8192 : 4096)

  config.libfoo.buffer = ($config.libfoo.buffer < $min_buffer \
                          ? $min_buffer                       \
                          : $config.libfoo.buffer)
}

accept ($config.libfoo.buffer >= ($config.libfoo.cache ? 8192 : 4096))
```

The interesting case to consider in the above example is when `config.libfoo.cache` changes from `true` to `false`: without the reset to defaults semantics the `prefer` clause would have kept the buffer at 8KB (since it's greater than the 4KB minimum).

Currently `accept` is always evaluated after `prefer` and temporary variables (like `min_buffer` in the above example) set in `prefer` are visible in `accept`. But it's best not to rely on this in case it changes in the future. For example, we may try harder to resolve the "yo-yo'ing" case mentioned above by checking if one of the alternating configurations are acceptable to everyone without re-evaluation.

This is also the reason why we need a separate `accept` in the first place. Plus, it allows for more advanced configuration techniques where we may need to have an acceptance criteria but no preferences.

Configuration variables that are set by the dependent in the `prefer` clause are visible in the subsequent clauses as well as in the subsequent `depends` values of this dependent. Configuration variables that are not set, however, are only visible until the immediately following `reflect` clause. For example, in the above listing, `config.libfoo.cache` would still be visible in the `reflect` clause if it were to follow `accept` but no further. As a result, if we need to make decisions based on configuration variables that we have no preference about, they need to be saved in the `reflect` clause. For example:

```
depends:
\
libfoo ^1.0.0
{
  # We have no preference about the cache but need to
  # observe its value.
  #
  prefer
  {
  }

  accept (true)

  reflect
  {
    config.hello.libfoo_cache = $config.libfoo.cache
  }
}
\

depends: libbar ^1.0.0 ? ($config.hello.libfoo_cache)
```

It is possible to determine the origin of the configuration variable value using the `$config.origin()` function. It returns either `undefined` if the variable is undefined (only possible if it has no default value), `default` if the variable has the default value from the `config` directive in `root.build`, `buildfile` if the value is from a `buildfile`, normally `config.build`, or `override` if the value is a command line override (that is, user configuration). For example, this is how we could use it if we only wanted to change the default value (notice that it's the variable's name and not its `$`-expansion that we pass to `$config.origin()`):

```
prefer
{
  config.libfoo.buffer = (                              \
    $config.origin(config.libfoo.buffer) == 'default' \
    ? 4096                                              \
    : $config.libfoo.buffer)
}
```

The following sub-sections discuss a number of more advanced configuration techniques that are based on the functionality described in this section.

5.1 Prefer X but Accept X or Y

Consider a configuration variable that is a choice between several mutually exclusive values, for example, user interface backends that could be, say, `cli`, `gui`, or `none`. In such situations it's common to prefer one value but being able to work with some subset of them. For example, we could prefer `gui` but were also able to make do with `cli` but not with `none`. Here is how we could express such a configuration:

```
libfoo ^1.0.0
{
  # We prefer `gui`, can also work with `cli` but not `none`.
  #
  prefer
  {
    config.libfoo.ui = (                                          \
      $config.origin(config.libfoo.ui) == 'default' ||            \
      ($config.libfoo.ui != 'gui' && $config.libfoo.ui != 'cli') \
      ? 'gui'                                                     \
      : $config.libfoo.ui)
  }

  accept ($config.libfoo.ui == 'gui' || $config.libfoo.ui == 'cli')
}
```

5.2 Use If Enabled

Sometimes we may want to use a feature if it is enabled by someone else but not enable it ourselves. For example, the feature might be expensive and our use of it tangential, but if it's enabled anyway, then we might as well take advantage of it. Here is how we could express such a configuration:

```
libfoo ^1.0.0
{
  # Use config.libfoo.x only if enabled by someone else.
  #
  prefer
  {
  }

  accept (true)

  reflect
```

```
  {
    config.hello.libfoo_x = $config.libfoo.x
  }
}
```

5.3 Disable If Enabled by Default

Sometimes we may want to disable a feature that is enabled by default provided that nobody else needs it. For example, the feature might be expensive and we would prefer to avoid paying the cost if we are the only ones using this dependency. Here is how we could express such a configuration:

```
libfoo ^1.0.0
{
  prefer
  {
    if ($config.origin(config.libfoo.x) == 'default')
      config.libfoo.x = false
  }

  accept (true)
}
```

6 Manifests

This chapter describes the general manifest file format as well as the concrete manifests used by `bpkg`.

Currently, three manifests are defined: package manifest, repository manifest, and signature manifest. The former two manifests can also be combined into a list of manifests to form the list of available packages and the description of a repository, respectively.

6.1 Manifest Format

A manifest is a UTF-8 encoded text restricted to the Unicode graphic characters, tabs (`\t`), carriage returns (`\r`), and line feeds (`\n`). It contains a list of name-value pairs in the form:

```
<name>: <value>
```

For example:

```
name: libfoo
version: 1.2.3
```

If a value needs to be able to contain other Unicode codepoints, they should be escaped in a value-specific manner. For example, the backslash (`\`) escaping described below can be extended for this purpose.

The name can contain any characters except `:` and whitespaces. Newline terminates the pair unless escaped with `\` (see below). Leading and trailing whitespaces before and after name and value are ignored except in the multi-line mode (see below).

If the first non-whitespace character on the line is `#`, then the rest of the line is treated as a comment and ignored except if the preceding newline was escaped or in the multi-line mode (see below). For example:

```
# This is a comment.
short: This is #not a comment
long: Also \
#not a comment
```

The first name-value pair in the manifest file should always have an empty name. The value of this special pair is the manifest format version. The version value shall use the default (that is, non-multi-line) mode and shall not use any escape sequences. Currently it should be `1`, for example:

```
: 1
name: libfoo
version: 1.2.3
```

Any new name that is added without incrementing the version must be optional so that it can be safely ignored by older implementations.

The special empty name pair can also be used to separate multiple manifests. In this case the version may be omitted in the subsequent manifests, for example:

```
: 1
name: libfoo
version: 1.2.3
:
name: libbar
version: 2.3.4
```

To disable treating of a newline as a name-value pair terminator we can escape it with `\`. Note that `\` is only treated as an escape sequence when followed by a newline and both are simply removed from the stream (as opposed to being replaced with a space). To enter a literal `\` at the end of the value, use the `\\` sequence. For example:

```
description: Long text that doesn't fit into one line \
so it is continued on the next line.

windows-path: C:\foo\bar\\
```

Notice that in the final example only the last `\` needs special handling since it is the only one that is followed by a newline.

One may notice that in this newline escaping scheme a line consisting of just `\` followed by a newline has no use, except, perhaps, for visual presentation of, arguably, dubious value. For example, this representation:

```
description: First line. \
\
Second line.
```

Is semantically equivalent to:

```
description: First line. Second line.
```

As a result, such a sequence is "overloaded" to provide more useful functionality in two ways: Firstly, if `:` after the name is followed on the next line by just `\` and a newline, then it signals the start of the multi-line mode. In this mode all subsequent newlines and `#` are treated as ordinary characters rather than value terminators or comments until a line consisting of just `\` and a newline (the multi-line mode terminator). For example:

```
description:
\
First paragraph.
#
Second paragraph.
\
```

Expressed as a C-string, the value in the above example is:

```
"First paragraph.\n#\nSecond paragraph."
```

Originally, the multi-line mode was entered if `:` after the name were immediately followed by `\` and a newline but on the same line. While this syntax is still recognized for backwards compatibility, it is deprecated and will be discontinued in the future.

Note that in the multi-line mode we can still use newline escaping to split long lines, for example:

```
description:
\
First paragraph that doesn't fit into one line \
so it is continued on the next line.
Second paragraph.
\
```

And secondly, in the simple (that is, non-multi-line) mode, the sole `\` and newline sequence is overloaded to mean a newline. So the previous example can also be represented like this:

```
description: First paragraph that doesn't fit into one \
line so it is continued on the next line.\
\
Second paragraph.
```

Note that the multi-line mode can be used to capture a value with leading and/or trailing whitespaces, for example:

```
description:
\
  test

\
```

The C-string representing this value is:

```
"  test\n"
```

EOF can be used instead of a newline to terminate both simple and multi-line values. For example the following representation results in the same value as in the previous example.

```
description:
\
  test

<EOF>
```

By convention, names are all in lower case and multi-word names are separated with −. Note that names are case-sensitive.

Also by convention, the following name suffixes are used to denote common types of values:

```
-file
-url
-email
```

For example:

```
description: Inline description
description-file: README
package-url: http://www.example.com
package-email: john@example.com
```

Other common name suffixes (such as -feed) could be added later.

Generally, unless there is a good reason not to, we keep values lower-case (for example, `requires` values such as `c++11` or `linux`). An example where we use upper/mixed case would be `license`; it seems unlikely `gplv2` would be better than `GPLv2`.

A number of name-value pairs described below allow for the value proper to be optionally followed by **`;`** and a comment. Such comments serve as additional documentation for the user and should be one or more full sentences, that is start with a capital letter and end with a period. Note that unlike #-style comments which are ignored, these comments are considered to be part of the value. For example:

```
email: foo-users@example.com ; Public mailing list.
```

It is recommended that you keep comments short, single-sentence. Note that non-comment semicolons in such values have to be escaped with a backslash, for example:

```
url: http://git.example.com/?p=foo\;a=tree
```

The only other recognized escape sequence in such values is `\\`, which is replaced with a single backslash. If a backslash is followed by any other character, then it is treated literally.

If a value with a comment is multi-line, then `;` must appear on a separate line, for example:

```
url:
\
http://git.example.com/?p=foo;a=tree
;
Git repository tree.
\
```

In this case, only lines that consist of a sole non-comment semicolon need escaping, for example:

```
license:
\
other: strange
\;
license
\
```

The only other recognized escape sequence in such multi-line values is lines consisting of two or more backslashes followed by a semicolon.

In the manifest specifications described below optional components are enclosed in square brackets (`[]`). If the name is enclosed in `[]` then the name-value pair is optional, otherwise – required. For example:

```
name: <name>
license: <licenses> [; <comment>]
[description]: <text>
```

In the above example `name` is required, `license` has an optional component (comment), and `description` is optional.

In certain situations (for example, shell scripts) it can be easier to parse the binary manifest representation. The binary representation does not include comments and consists of a sequence of name-value pairs in the following form:

```
<name>:<value>\0
```

That is, the name and the value are separated by a colon and each pair (including the last) is terminated with the `NUL` character. Note that there can be no leading or trailing whitespace characters around the name and any whitespaces after the colon and before the `NUL` terminator are part of the value. Finally, the manifest format versions are always explicit (that is, not empty) in binary manifest lists.

6.2 Package Manifest

The package manifest (the `manifest` file found in the package's root directory) describes a `bpkg` package. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

The subset of the values up to and including `license` constitute the package manifest header. Note that the header is a valid package manifest since all the other values are optional. There is also no requirement for the header values to appear first or to be in a specific order. In particular, in a full package manifest they can be interleaved with non-header values.

```
name: <name>
version: <version>
[project]: <name>
[priority]: <priority> [; <comment>]
summary: <text>
license: <licenses> [; <comment>]

[topics]: <topics>
[keywords]: <keywords>
[description]: <text>
[description-file]: <path> [; <comment>]
[description-type]: <text-type>
[changes]: <text>
[changes-file]: <path> [; <comment>]

[url]: <url> [; <comment>]
[doc-url]: <url> [; <comment>]
[src-url]: <url> [; <comment>]
[package-url]: <url> [; <comment>]

[email]: <email> [; <comment>]
[package-email]: <email> [; <comment>]
[build-email]: <email> [; <comment>]
[build-warning-email]: <email> [; <comment>]
[build-error-email]: <email> [; <comment>]

[depends]: [*] <alternatives> [; <comment>]
[requires]: [*] <alternatives> [; <comment>]

[tests]: [*] <name> [<version-constraint>]
[examples]: [*] <name> [<version-constraint>]
[benchmarks]: [*] <name> [<version-constraint>]

[builds]: <class-expr> [; <comment>]
[build-include]: <config>[/<target>] [; <comment>]
[build-exclude]: <config>[/<target>] [; <comment>]

[build-file]: <path>

[bootstrap-build]: <text>
[root-build]: <text>
[*-build]: <text>

[bootstrap-build2]: <text>
[root-build2]: <text>
[*-build2]: <text>
```

6.2.1 name

```
name: <name>
```

The package name. See Package Name for the package name format description. Note that the name case is preserved for display, in file names, etc.

6.2.2 `version`

```
version: <version>
[upstream-version]: <string>
```

The package version. See Package Version for the version format description. Note that the version case is preserved for display, in file names, etc.

When packaging existing projects, sometimes you may want to deviate from the upstream versioning scheme because, for example, it may not be representable as a `bpkg` package version or simply be inconvenient to work with. In this case you would need to come up with an upstream-to-downstream version mapping and use the `upstream-version` value to preserve the original version for information.

6.2.3 `project`

```
[project]: <name>
```

The project this package belongs to. The project name has the same restrictions as the package name (see Package Name for details) and its case is preserved for display, in directory names, etc. If unspecified, then the project name is assumed to be the same as the package name.

Projects are used to group related packages together in order to help with organization and discovery in repositories. For example, packages `hello`, `libhello`, and `libhello2` could all belong to project `hello`. By convention, projects of library packages are named without the `lib` prefix.

6.2.4 `priority`

```
[priority]: <priority> [; <comment>]

<priority> = security | high | medium | low
```

The release priority (optional). As a guideline, use `security` for security fixes, `high` for critical bug fixes, `medium` for important bug fixes, and `low` for minor fixes and/or feature releases. If not specified, `low` is assumed.

6.2.5 `summary`

```
summary: <text>
```

The short description of the package.

6.2.6 `license`

```
license: <licenses> [; <comment>]

<licenses> = <license> [, <license>]*
<license>  = [<scheme>:] <name>
<scheme>   = other
```

The package license. The default license name scheme is SPDX License Expression. In its simplest form, it is just an ID of the license under which this package is distributed. An optional comment normally gives the full name of the license, for example:

```
license: MPL-2.0 ; Mozilla Public License 2.0
```

The following table lists the most commonly used free/open source software licenses and their SPDX license IDs:

```
MIT                ; MIT License.

BSD-2-Clause       ; BSD 2-Clause "Simplified" License
BSD-3-Clause       ; BSD 3-Clause "New" or "Revised" License
BSD-4-Clause       ; BSD 4-Clause "Original" or "Old" License

GPL-2.0-only       ; GNU General Public License v2.0 only
GPL-2.0-or-later   ; GNU General Public License v2.0 or later
GPL-3.0-only       ; GNU General Public License v3.0 only
GPL-3.0-or-later   ; GNU General Public License v3.0 or later

LGPL-2.0-only      ; GNU Library General Public License v2 only
LGPL-2.0-or-later  ; GNU Library General Public License v2 or later
LGPL-2.1-only      ; GNU Lesser General Public License v2.1 only
LGPL-2.1-or-later  ; GNU Lesser General Public License v2.1 or later
LGPL-3.0-only      ; GNU Lesser General Public License v3.0 only
LGPL-3.0-or-later  ; GNU Lesser General Public License v3.0 or later

AGPL-3.0-only      ; GNU Affero General Public License v3.0 only
AGPL-3.0-or-later  ; GNU Affero General Public License v3.0 or later

Apache-1.0         ; Apache License 1.0
Apache-1.1         ; Apache License 1.1
Apache-2.0         ; Apache License 2.0

MPL-1.0            ; Mozilla Public License 1.0
MPL-1.1            ; Mozilla Public License 1.1
MPL-2.0            ; Mozilla Public License 2.0

BSL-1.0            ; Boost Software License 1.0

Unlicense          ; The Unlicense (public domain)
```

If the package is licensed under multiple licenses, then an SPDX license expression can be used to specify this, for example:

```
license: Apache-2.0 OR MIT
license: MIT AND BSD-2-Clause
```

A custom license or extra conditions can be expressed either using the license reference mechanism of the SPDX license expression or using the `other` scheme (described below). For example:

```
license: LicenseRef-My-MIT-Like; Custom MIT-alike license
license: other: MIT with extra attribution requirements
```

The `other` license name scheme can be used to specify licenses that are not defined by SPDX. The license names in this scheme are free form with case-insensitive comparison. The following names in this scheme have predefined meaning:

```
other: public domain    ; Released into the public domain
other: available source ; Not free/open source with public source code
other: proprietary      ; Not free/open source
other: TODO             ; License is not yet decided
```

For new projects The Unlicense disclaimer with the `Unlicense` SPDX ID is recommended over `other: public domain`.

To support combining license names that use different schemes, the `license` manifest value can contain a comma-separated list of license names. This list has the *AND* semantics, that is, the user must comply with all the licenses listed. To capture alternative licensing options (the *OR* semantics), multiple `license` manifest values are used, for example:

```
license: GPL-2.0-only, other: available source
license: other: proprietary
```

For complex licensing situations it is recommended to add comments as an aid to the user, for example:

```
license: LGPL-2.1-only AND MIT ; If linking with GNU TLS.
license: BSD-3-Clause          ; If linking with OpenSSL.
```

For backwards compatibility with existing packages, the following (deprecated) scheme-less values on the left are recognized as aliases for the new values on the right:

```
BSD2               BSD-2-Clause
BSD3               BSD-3-Clause
BSD4               BSD-4-Clause
GPLv2              GPL-2.0-only
GPLv3              GPL-3.0-only
LGPLv2             LGPL-2.0-only
LGPLv2.1           LGPL-2.1-only
LGPLv3             LGPL-3.0-only
AGPLv3             AGPL-3.0-only
ASLv1              Apache-1.0
ASLv1.1            Apache-1.1
ASLv2              Apache-2.0
MPLv2              MPL-2.0

public domain      other: public domain
available source   other: available source
proprietary        other: proprietary
TODO               other: TODO
```

6.2.7 `topics`

```
[topics]: <topics>

<topics> = <topic> [, <topic>]*
```

The package topics (optional). The format is a comma-separated list of up to five potentially multi-word concepts that describe this package. For example:

```
topics: xml parser, xml serializer
```

6.2.8 `keywords`

```
[keywords]: <keywords>

<keywords> = <keyword> [ <keyword>]*
```

The package keywords (optional). The format is a space-separated list of up to five words that describe this package. Note that the package and project names as well as words from its summary are already considered to be keywords and need not be repeated in this value.

6.2.9 `description`

```
[description]: <text>
[description-file]: <path> [; <comment>]
[description-type]: <text-type>
```

The detailed description of the package. It can be provided either inline as a text fragment or by referring to a file within a package (e.g., `README`), but not both.

In the web interface (`brep`) the description is displayed according to its type. Currently, pre-formatted plain text, GitHub-Flavored Markdown, and CommonMark are supported with the following `description-type` values, respectively:

```
text/plain
text/markdown;variant=GFM
text/markdown;variant=CommonMark
```

If just `text/markdown` is specified, then the GitHub-Flavored Markdown (which is a superset of CommonMark) is assumed.

If the description type is not explicitly specified and the description is specified as `description-file`, then an attempt to derive the type from the file extension is made. Specifically, the **`.md`** and **`.markdown`** extensions are mapped to `text/markdown`, the **`.txt`** and no extension are mapped to `text/plain`, and all other extensions are treated as an unknown type, similar to unknown `description-type` values. And if the description is not specified as a file, `text/plain` is assumed.

6.2.10 `changes`

```
[changes]: <text>
[changes-file]: <path> [; <comment>]
```

The description of changes in the release.

The tricky aspect is what happens if the upstream release stays the same (and has, say, a `NEWS` file to which we point) but we need to make another package release, for example, to apply a critical patch.

Multiple `changes` values can be present which are all concatenated in the order specified, that is, the first value is considered to be the most recent (similar to `ChangeLog` and `NEWS` files). For example:

```
changes: 1.2.3-2: applied upstream patch for critical bug bar
changes: 1.2.3-1: applied upstream patch for critical bug foo
changes-file: NEWS
```

Or:

```
changes:
\
1.2.3-2
  - applied upstream patch for critical bug bar
  - regenerated documentation

1.2.3-1
  - applied upstream patch for critical bug foo
\
changes-file: NEWS
```

In the web interface (`brep`) the changes are displayed as pre-formatted plain text, similar to the package description.

6.2.11 `url`

```
[url]: <url> [; <comment>]
```

The project home page URL.

6.2.12 `doc-url`

```
[doc-url]: <url> [; <comment>]
```

The project documentation URL.

6.2.13 `src-url`

```
[src-url]: <url> [; <comment>]
```

The project source repository URL.

6.2.14 `package-url`

```
[package-url]: <url> [; <comment>]
```

The package home page URL. If not specified, then assumed to be the same as `url`. It only makes sense to specify this value if the project and package are maintained separately.

6.2.15 `email`

```
[email]: <email> [; <comment>]
```

The project email address. For example, a support mailing list.

6.2.16 `package-email`

```
[package-email]: <email> [; <comment>]
```

The package email address. If not specified, then assumed to be the same as `email`. It only makes sense to specify this value if the project and package are maintained separately.

6.2.17 `build-email`

```
[build-email]: <email> [; <comment>]
```

The build notification email address. It is used to send build result notifications by automated build bots. If unspecified, then no build result notifications for this package are sent by email.

For backwards compatibility with existing packages, if it is specified but empty, then this is the same as unspecified.

6.2.18 `build-warning-email`

```
[build-warning-email]: <email> [; <comment>]
```

The build warning notification email address. Unlike `build-email`, only build warning and error notifications are sent to this email.

6.2.19 `build-error-email`

```
[build-error-email]: <email> [; <comment>]
```

The build error notification email address. Unlike `build-email`, only build error notifications are sent to this email.

6.2.20 depends

```
[depends]: [*] <alternatives> [; <comment>]
```

Single-line form:

```
<alternatives> = <alternative> [ '|' <alternative>]*
<alternative>  = <dependencies> ['?' <enable-cond>] [<reflect-var>]
<dependencies> = <dependency> | \
    '{' <dependency> [<dependency>]* '}' [<version-constraint>]
<dependency>   = <name> [<version-constraint>]
<enable-cond>  = '(' <buildfile-eval-expr> ')'
<reflect-var>  = <config-var> '=' <value>
```

Multi-line form:

```
<alternatives> =
  <alternative>[
  '|'
  <alternative>]*

<alternative>  =
  <dependencies>
  '{'
   [
     'enable' <enable-cond>
   ]

   [
     'require'
     '{'
        <buildfile-fragment>
     '}'

   ] | [

     'prefer'
     '{'
        <buildfile-fragment>
     '}'

     'accept' <accept-cond>
   ]

   [
     'reflect'
     '{'
        <buildfile-fragment>
     '}'
   ]
  '}'

<accept-cond>  = '(' <buildfile-eval-expr> ')'
```

The dependency packages. The most common form of a dependency is a package name followed by the optional version constraint. For example:

```
depends: libhello ^1.0.0
```

See Package Version Constraint for the format and semantics of the version constraint. Instead of a concrete value, the version in the constraint can also be specified in terms of the dependent package's version (that is, its `version` value) using the special `$` value. This mechanism is primarily useful when developing related packages that should track each other's versions exactly or closely. For example:

```
name: sqlite3
version: 3.18.2
depends: libsqlite3 == $
```

If multiple packages are specified within a single `depends` value, they must be grouped with `{}`. This can be useful if the packages share a version constraint. The group constraint applies to all the packages in the group that do not have their own constraint. For example:

```
depends: { libboost-any libboost-log libboost-uuid ~1.77.1 } ~1.77.0
```

If the `depends` value starts with `*`, then it is a *build-time* dependency. Otherwise it is *run-time*. For example:

```
depends: * byacc >= 20210619
```

Most of the build-time dependencies are expected to be tools such as code generators, so you can think of `*` as the executable mark printed by `ls`. An important difference between the two kinds of dependencies is that in case of cross-compilation a build-time dependency must be built for the host machine, not the target. Build system modules are also build-time dependencies.

Two special build-time dependency names are recognized and checked in an ad hoc manner: `build2` (the `build2` build system) and `bpkg` (the `build2` package manager). This allows us to specify the minimum required build system and package manager versions, for example:

```
depends: * build2 >= 0.15.0
depends: * bpkg >= 0.15.0
```

If you are developing or packaging a project that uses features from the not yet released (staged) version of the `build2` toolchain, then you can use the pre-release version in the constraint. For example:

```
depends: * build2 >= 0.16.0-
depends: * bpkg >= 0.16.0-
```

A dependency can be conditional, that is, it is only enabled if a certain condition is met. For example:

```
depends: libposix-getopt ^1.0.0 ? ($cxx.target.class == 'windows')
```

The condition after `?` inside `()` is a `buildfile` eval context expression that should evaluate to `true` or `false`, as if it were specified in the `buildfile if` directive (see Expansion and Quoting and Conditions (`if-else`) for details).

The condition expression is evaluated after loading the package build system skeleton, that is, after loading its `root.build` (see Package Build System Skeleton for details). As a result, variable values set by build system modules that are loaded in `root.build` as well as the package's configuration (including previously reflected; see below) or computed values can be referenced in dependency conditions. For example, given the following `root.build`:

```
# root.build

...

using cxx

# MinGW ships POSIX <getopt.h>.
#
need_getopt = ($cxx.target.class == 'windows' && \
               $cxx.target.system != 'mingw32')

config [bool] config.hello.regex ?= false
```

We could have the following conditional dependencies:

```
depends: libposix-getopt ^1.0.0 ? ($need_getopt) ; Windows && !MinGW.
depends: libposix-regex  ^1.0.0 ? ($config.hello.regex && \
                                   $cxx.target.class == 'windows')
```

The first `depends` value in the above example also shows the use of an optional comment. It's a good idea to provide it if the condition is not sufficiently self-explanatory.

A dependency can "reflect" configuration variables to the subsequent `depends` values and to the package configuration. This can be used to signal whether a conditional dependency is enabled or which dependency alternative was selected (see below). The single-line form of `depends` can only reflect one configuration variable. For example:

```
depends: libposix-regex ^1.0.0       \
  ? ($cxx.target.class == 'windows') \
  config.hello.external_regex=true

# root.build

...

using cxx

config [bool] config.hello.external_regex ?= false

# buildfile

libs =

if $config.hello.external_regex
  import libs += libposix-regex%lib{posix-regex}

exe{hello}: ... $libs
```

In the above example, if the `hello` package is built for Windows, then the dependency on `libposix-regex` will be enabled and the package will be configured with `config.hello.external_regex=true`. This is used in the `buildfile` to decide whether to import `libposix-regex`. While in this example it would have probably been easier to just duplicate the check for Windows in the `buildfile` (or, better yet, factor this check to `root.build` and share the result via a computed variable between `manifest` and `buildfile`), the reflect mechanism is the only way to communicate the selected dependency alternative (discussed next).

An attempt to set a reflected configuration variable that is overridden by the user is an error. In a sense, configuration variables that are used to reflect information should be treated as the package's implementation details if the package management is involved. If, however, the package is configured without `bpkg`'s involvement, then these variables could reasonably be provided as user configuration.

If you feel the need to allow a reflected configuration variable to also potentially be supplied as user configuration, then it's probably a good sign that you should turn things around: make the variable only user-configurable and use the enable condition instead of reflect. Alternatively, you could try to recognize and handle user overrides with the help of the `$config.origin()` function discussed in Dependency Configuration Negotiation.

While multiple `depends` values are used to specify multiple packages with the *AND* semantics, inside `depends` we can specify multiple packages (or groups of packages) with the *OR* semantics, called dependency alternatives. For example:

```
depends: libmysqlclient >= 5.0.3 | libmariadb ^10.2.2
```

When selecting an alternative, `bpkg` only considers packages that are either already present in the build configuration or are selected as dependencies by other packages, picking the first alternative with a satisfactory version constraint and an acceptable configuration. As a result, the order of alternatives expresses a preference. If, however, this does not yield a suitable alternative, then `bpkg` fails asking the user to make the selection.

For example, if the package with the above dependency is called `libhello` and we build it in a configuration that already has both `libmysqlclient` and `libmariadb`, then `bpkg` will select `libmysqlclient`, provided the existing version satisfies the version constraint. If, however, there are no existing packages in the build configuration and we attempt to build just `libhello`, then `bpkg` will fail asking the user to pick one of the alternatives. If we wanted to make `bpkg` select `libmariadb` we could run:

```
$ bpkg build libhello ?libmariadb
```

While `bpkg`'s refusal to automatically pick an alternative that would require building a new package may at first seem unfriendly to the user, practical experience shows that such extra user-friendliness would rarely justify the potential confusion that it may cause.

Also note that it's not only the user that can pick a certain alternative but also a dependent package. Continuing with the above example, if we had `hello` that depended on `libhello` but only supported MariaDB (or provided a configuration variable to explicitly select the database), then we could have the following in its `manifest`:

```
depends: libmariadb       ; Select MariaDB in libhello.
depends: libhello ^1.0.0
```

Dependency alternatives can be combined with all the other features discussed above: groups, conditional dependencies, and reflect. As mentioned earlier, reflect is the only way to communicate the selection to subsequent `depends` values and the package configuration. For example:

```
depends: libmysqlclient >= 5.0.3 config.hello.db='mysql'      | \
         libmariadb ^10.2.2 ? ($cxx.target.class != 'windows')  \
           config.hello.db='mariadb'

depends: libz ^1.2.1100 ? ($config.hello.db == 'mysql')
```

If an alternative is conditional and the condition evaluates to `false`, then this alternative is not considered. If all but one alternative are disabled due to conditions, then this becomes an ordinary dependency. If all the alternatives are disabled due to conditions, then the entire dependency is disabled. For example:

```
depends: libmysqlclient >= 5.0.3 ? ($config.hello.db == 'mysql') | \
         libmariadb ^10.2.2      ? ($config.hello.db == 'mariadb')
```

While there is no need to use the dependency alternatives in the above example (since the alternatives are mutually exclusive), it makes for good documentation of intent.

Besides as a single line, the `depends` value can also be specified in a multi-line form which, besides potentially better readability, provides additional functionality. In the multi-line form, each dependency alternative occupies a separate line and `|` can be specified either at the end of the dependency alternative line or on a separate line. For example:

```
depends:
\
libmysqlclient >= 5.0.3 ? ($config.hello.db == 'mysql') |
libmariadb ^10.2.2      ? ($config.hello.db == 'mariadb')
\
```

A dependency alternative can be optionally followed by a block containing a number of clauses. The `enable` clause is the alternative way to specify the condition for a conditional dependency while the `reflect` clause is the alternative way to specify the reflected configuration variable. The block may also contain #-style comments, similar to `buildfile`. For example:

```
depends:
\
libmysqlclient >= 5.0.3
{
  reflect
  {
```

```
    config.hello.db = 'mysql'
  }
}
|
libmariadb ^10.2.2
{
  # TODO: MariaDB support on Windows.
  #
  enable ($cxx.target.class != 'windows')

  reflect
  {
    config.hello.db = 'mariadb'
  }
}
\
```

While the `enable` clause is essentially the same as its inline `?` variant, the `reflect` clause is an arbitrary `buildfile` fragment that can have more complex logic and assign multiple configuration variables. For example:

```
libmariadb ^10.2.2
{
  reflect
  {
    if ($cxx.target.class == 'windows')
      config.hello.db = 'mariadb-windows'
    else
      config.hello.db = 'mariadb-posix'
  }
}
```

The multi-line form also allows us to express our preferences and requirements for the dependency configuration. If all we need is to set one or more `bool` configuration variables to `true` (which usually translates to enabling one or more features), then we can use the `require` clause. For example:

```
libmariadb ^10.2.2
{
  require
  {
    config.libmariadb.cache = true

    if ($cxx.target.class != 'windows')
      config.libmariadb.tls = true
  }
}
```

For more complex dependency configurations instead of `require` we can use the `prefer` and `accept` clauses. The `prefer` clause can set configuration variables of any type and to any value in order to express the package's preferred configuration while the `accept` condition evaluates whether any given configuration is acceptable. If used instead of `require`, both `prefer` and `accept` must be present. For example:

```
libmariadb ^10.2.2
{
  # We prefer the cache but can work without it.
  # We need the buffer of at least 4KB.
  #
  prefer
  {
    config.libmariadb.cache = true

    config.libmariadb.buffer = ($config.libmariadb.buffer < 4096 \
                                ? 4096                           \
                                : $config.libmariadb.buffer)
  }

  accept ($config.libmariadb.buffer >= 4096)
}
```

The `require` clause is essentially a shortcut for specifying the `prefer`/`accept` clauses where the `accept` condition simply verifies all the variable values assigned in the `prefer` clause. It is, however, further restricted to the common case of only setting `bool` variables and only to `true` to allow additional optimizations during the configuration negotiation.

The `require` and `prefer` clauses are arbitrary `buildfile` fragments similar to `reflect` while the `accept` clause is a `buildfile` eval context expression that should evaluate to `true` or `false`, similar to `enable`.

Given the `require` and `prefer`/`accept` clauses of all the dependents of a particular dependency, `bpkg` tries to negotiate a configuration acceptable to all of them as described in Dependency Configuration Negotiation.

All the clauses are evaluated in the specified order, that is, `enable`, then `require` or `prefer`/`accept`, and finally `reflect`, with the (negotiated, in case of `prefer`) configuration values set by preceding clauses available for examination by the subsequent clauses in this `depends` value as well as in all the subsequent ones. For example:

```
depends:
\
libmariadb ^10.2.2
{
  prefer
  {
    config.libmariadb.cache = true

    config.libmariadb.buffer = ($config.libmariadb.buffer < 4096 \
                                ? 4096                           \
                                : $config.libmariadb.buffer)
  }

  accept ($config.libmariadb.buffer >= 4096)

  reflect
  {
    config.hello.buffer = $config.libmariadb.buffer
  }
```

```
}
\

depends: liblru ^1.0.0 ? ($config.libmariadb.cache)
```

The above example also highlights the difference between the `require`/`prefer` and `reflect` clauses that is easy to mix up: in `require`/`prefer` we set the dependency's while in `reflect` we set the dependent's configuration variables.

6.2.21 `requires`

```
[requires]: [*] <alternatives> [; <comment>]

<alternatives> = <alternative> [ '|' <alternative>]*
<alternative>  = <requirements> ['?' [<enable-cond>]] [<reflect-var>]
<requirements> = [<requirement>] | \
    '{' <requirement> [<requirement>]* '}' [<version-constraint>]
<requirement>  = <name> [<version-constraint>]
<enable-cond>  = '(' <buildfile-eval-expr> ')'
<reflect-var>  = <config-var> '=' <value>
```

The package requirements other than other packages. Such requirements are normally checked in an ad hoc way during package configuration by its `buildfiles` and the primary purpose of capturing them in the manifest is for documentation. However, there are some special requirements that are recognized by the tooling (see below). For example:

```
requires: c++11
requires: linux | windows | macos
requires: libc++ ? ($macos) ; libc++ if using Clang on Mac OS.
```

The format of the `requires` value is similar to `depends` with the following differences. The requirement name (with or without version constraint) can mean anything (but must still be a valid package name). Only the `enable` and `reflect` clauses are permitted. There is a simplified syntax with either the requirement or enable condition or both being empty and where the comment carries all the information (and is thus mandatory). For example:

```
requires: ; X11 libs.
requires: ? ($windows) ; Only 64-bit.
requires: ? ; Only 64-bit if on Windows.
requires: x86_64 ? ; Only if on Windows.
```

Note that `requires` can also be used to specify dependencies on system libraries, that is, the ones not to be packaged. In this case it may make sense to also specify the version constraint. For example:

```
requires: libx11 >= 1.7.2
```

To assist potential future automated processing, the following pre-defined requirement names should be used for the common requirements:

```
c++98
c++03
c++11
c++14
c++17
c++20
c++23

posix
linux
macos
freebsd
openbsd
netbsd
windows

gcc[_X.Y.Z]  ; For example: gcc_6, gcc_4.9, gcc_5.0.0
clang[_X.Y]  ; For example: clang_6, clang_3.4, clang_3.4.1
msvc[_N.U]   ; For example: msvc_14, msvc_15.3
```

The following pre-defined requirement names are recognized by automated build bots:

```
bootstrap
host
```

The `bootstrap` value should be used to mark build system modules that require bootstrapping. The `host` value should be used to mark packages, such source code generators, that are normally specified as build-time dependencies by other packages and therefore should be built in a host configuration. See the `bbot` documentation for details.

6.2.22 `tests, examples, benchmarks`

```
[tests]: [*] <name> [<version-constraint>]
[examples]: [*] <name> [<version-constraint>]
[benchmarks]: [*] <name> [<version-constraint>]
```

Separate tests, examples, and benchmarks packages. If the value starts with `*`, then the primary package is a *build-time* dependency for the specified package. Otherwise it is *run-time*. See the `depends` value for details on *build-time* dependencies.

These packages are built and tested by automated build bots together with the primary package (see the `bbot` documentation for details). This, in particular, implies that these packages must be available from the primary package's repository or its complement repositories, recursively. The recommended naming convention for these packages is the primary package name followed by `-tests`, `-examples`, or `-benchmarks`, respectively. For example:

```
name: hello
tests : hello-tests
examples: hello-examples
```

See Package Version Constraint for the format and semantics of the optional version constraint. Instead of a concrete value, it can also be specified in terms of the primary package's version (see the `depends` value for details), for example:

```
tests: hello-tests ~$
```

Note that normally the tests, etc., packages themselves do not have an explicit dependency on the primary package (in a sense, the primary package has a special dependency on them). They are also not built by automated build bots separately from their primary package but may have their own build constraints, for example, to be excluded from building on some platforms where the primary package is still built, for example:

```
name: hello-tests
builds: -windows
```

6.2.23 `builds`

```
[builds]: [<class-uset> ':' ] [<class-expr>] [; <comment>]

<class-uset> = <class-name> [ <class-name>]*
<class-expr> = <class-term> [ <class-term>]*
<class-term> = ('+'|'-'|'&')['!'](<class-name> | '(' <class-expr> ')')
```

The package build configurations. They specify the build configuration classes the package should or should not be built for by automated build bots. For example:

```
builds: -windows
```

Build configurations can belong to multiple classes with their names and semantics varying between different build bot deployments. However, the pre-defined `none`, `default`, `all`, `host`, and `build2` classes are always provided. If no `builds` value is specified in the package manifest, then the `default` class is assumed.

A build configuration class can also derive from another class in which case configurations that belong to the derived class are treated as also belonging to the base class (or classes, recursively). See the Build Configurations page of the build bot deployment for the list of available build configurations and their classes.

The `builds` value consists of an optional underlying class set (`<class-uset>`) followed by a class set expression (`<class-expr>`). The underlying set is a space-separated list of class names that define the set of build configurations to consider. If not specified, then all the configurations belonging to the `default` class are assumed. The class set expression can then be used to exclude certain configurations from this initial set.

The class expression is a space-separated list of terms that are evaluated from left to right. The first character of each term determines whether the build configuration that belong to its set are added to (`+`), subtracted from (`-`), or intersected with (`&`) the current set. If the second character in the term is `!`, then its set of configuration is inverted against the underlying set. The term itself can be either the class name or a parenthesized expression. Some examples (based on the cppget.org deployment):

```
builds: none                    ; None.
builds: all                     ; All (suitable for libraries).
builds: host                    ; All host (suitable for tools).
builds: default                 ; All default.
```

```
builds: host : &default             ; Host default.
builds: default legacy              ; All default and legacy.
builds: host: &( +default +legacy ) ; Host default and legacy.
builds: -windows                    ; Default except Windows.
builds: all : -windows              ; All except Windows.
builds: all : -mobile               ; All except mobile.
builds: all : &gcc                  ; All with GCC only.
builds: all : &gcc-8+               ; All with GCC 8 and up only.
builds: gcc : -optimized            ; GCC without optimization.
builds: gcc : &( +linux +macos )    ; GCC on Linux and Mac OS.
```

Notice that the colon and parentheses must be separated with spaces from both preceding and following terms.

Multiple `builds` values are evaluated in the order specified and as if they were all part of a single expression. Only the first value may specify the underlying set. The main reason for having multiple values is to provide individual reasons (as the `builds` value comments) for different parts of the expression. For example:

```
builds: default experimental ; Only modern compilers are supported.
builds: -gcc                 ; GCC is not supported.
builds: -clang               ; Clang is not supported.

builds: default
builds: -( +macos &gcc)      ; Homebrew GCC is not supported.
```

The `builds` value comments are used by the web interface (`brep`) to display the reason for the build configuration exclusion.

After evaluating all the `builds` values, the final configuration set can be further fine-tuned using the `build-{include, exclude}` patterns.

6.2.24 `build-{include, exclude}`

```
[build-include]: <config>[/<target>] [; <comment>]
[build-exclude]: <config>[/<target>] [; <comment>]
```

The package build inclusions and exclusions. The `build-include` and `build-exclude` values further reduce the configuration set produced by evaluating the `builds` values. The *config* and *target* values are filesystem wildcard patterns which are matched against the build configuration names and target names (see the `bbot` documentation for details). In particular, the `*` wildcard matches zero or more characters within the name component while the `**` sequence matches across the components. Plus, wildcard-only pattern components match absent name components. For example:

```
build-exclude: windows**     # matches windows_10-msvc_15
build-exclude: macos*-gcc**  # matches macos_10.13-gcc_8.1-O3
build-exclude: linux-gcc*-*  # matches linux-gcc_8.1 and linux-gcc_8.1-O3
```

The exclusion and inclusion patterns are applied in the order specified with the first match determining whether the package will be built for this configuration and target. If none of the patterns match (or none we specified), then the package is built.

As an example, the following value will exclude 32-bit builds for the MSVC 14 compiler:

```
build-exclude: *-msvc_14**/i?86-** ; Linker crash.
```

As another example, the following pair of values will make sure that a package is only built on Linux:

```
build-include: linux**
build-exclude: ** ; Only supported on Linux.
```

Note that the comment of the matching exclusion is used by the web interface (`brep`) to display the reason for the build configuration exclusion.

6.2.25 `build-file`

```
[build-file]: <path>

[bootstrap-build]: <text>
[root-build]: <text>
[*-build]: <text>

[bootstrap-build2]: <text>
[root-build2]: <text>
[*-build2]: <text>
```

The contents of the mandatory `bootstrap.build` file, optional `root.build` file, and additional files included by `root.build`, or their alternative naming scheme variants (`bootstrap.build2`, etc). Packages with the alternative naming scheme should use the `*-build2` values instead of `*-build`. See Package Build System Skeleton for background.

These files must reside in the package's `build/` subdirectory and have the `.build` extension (or their alternative names). They can be provided either inline as text fragments or, for additional files, by referring to them with a path relative to this subdirectory, but not both. The `*-build`/`*-build2` manifest value name prefixes must be the file paths relative to this subdirectory with the extension stripped.

As an example, the following values correspond to the `build/config/common.build` file:

```
build-file: config/common.build

config/common-build:
\
config [bool] config.libhello.fancy ?= false
\
```

And the following values correspond to the `build2/config/common.build2` file in a package with the alternative naming scheme:

```
build-file: config/common.build2

config/common-build2:
\
config [bool] config.libhello.fancy ?= false
\
```

If unspecified, then the package's `bootstrap.build`, `root.build`, and `build/config/*.build` files (or their alternative names) will be automatically added, for example, when the package list manifest is created.

6.3 Package List Manifest for `pkg` Repositories

The package list manifest (the `packages.manifest` file found in the **`pkg`** repository root directory) describes the list of packages available in the repository. First comes a manifest that describes the list itself (referred to as the list manifest). The list manifest synopsis is presented next:

```
sha256sum: <sum>
```

After the list manifest comes a (potentially empty) sequence of package manifests. These manifests shall not contain any `*-file` or incomplete `depends` values (such values should be converted to their inline versions or completed, respectively) but must contain the `*-build` values (unless the corresponding files are absent) and the following additional (to package manifest) values:

```
location: <path>
sha256sum: <sum>
```

The detailed description of each value follows in the subsequent sections.

6.3.1 `sha256sum` (list manifest)

```
sha256sum: <sum>
```

The SHA256 checksum of the `repositories.manifest` file (described below) that corresponds to this repository. The *sum* value should be 64 characters long (that is, just the SHA256 value, no file name or any other markers), be calculated in the binary mode, and use lower-case letters.

This checksum is used to make sure that the `repositories.manifest` file that was fetched is the same as the one that was used to create the `packages.manifest` file. This also means that if `repositories.manifest` is modified in any way, then `packages.manifest` must be regenerated as well.

6.3.2 `location` (package manifest)

```
location: <path>
```

The path to the package archive file relative to the repository root. It should be in the POSIX representation.

if the repository keeps multiple versions of the package and places them all into the repository root directory, it can get untidy. With `location` we allow for sub-directories.

6.3.3 `sha256sum` (package manifest)

```
sha256sum: <sum>
```

The SHA256 checksum of the package archive file. The *sum* value should be 64 characters long (that is, just the SHA256 value, no file name or any other markers), be calculated in the binary mode, and use lower-case letters.

6.4 Package List Manifest for `dir` Repositories

The package list manifest (the `packages.manifest` file found in the **`dir`** repository root directory) describes the list of packages available in the repository. It is a (potentially empty) sequence of manifests with the following synopsis:

```
location: <path>
[fragment]: <string>
```

The detailed description of each value follows in the subsequent sections. The `fragment` value can only be present in a merged `packages.manifest` file for a multi-fragment repository.

As an example, if our repository contained the `src/` subdirectory that in turn contained the `libfoo` and `foo` packages, then the corresponding `packages.manifest` file could look like this:

```
: 1
location: src/libfoo/
:
location: src/foo/
```

6.4.1 `location`

```
location: <path>
```

The path to the package directory relative to the repository root. It should be in the POSIX representation.

6.4.2 `fragment`

```
[fragment]: <string>
```

The repository fragment id this package belongs to.

6.5 Repository Manifest

The repository manifest (only used as part of the repository manifest list described below) describes a **`pkg`**, **`dir`**, or **`git`** repository. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
[location]: <uri>
[type]: pkg|dir|git
[role]: base|prerequisite|complement
[trust]: <fingerprint>
[url]: <url>
[email]: <email> [; <comment>]
[summary]: <text>
[description]: <text>
[certificate]: <pem>
[fragment]: <string>
```

See also the Repository Chaining documentation for further information @@ TODO.

6.5.1 `location`

```
[location]: <uri>
```

The repository location. The location can and must only be omitted for the base repository. Since we got hold of its manifest, then we presumably already know the location of the base repository. If the location is a relative path, then it is treated as relative to the base repository location.

For the **`git`** repository type the relative location does not inherit the URL fragment from the base repository. Note also that the remote **`git`** repository locations normally have the **`.git`** extension that is stripped when a repository is cloned locally. To make the relative locations usable in both contexts, the **`.git`** extension should be ignored if the local prerequisite repository with the extension does not exist while the one without the extension does.

While POSIX systems normally only support POSIX paths (that is, forward slashes only), Windows is generally able to handle both slash types. As a result, it is recommended that POSIX paths are always used in the `location` values, except, perhaps, if the repository is explicitly Windows-only by, for example, having a location that is an absolute Windows path with the drive letter. The **`bpkg`** package manager will always try to represent the location as a POSIX path and only fallback to the native representation if that is not possible (for example, there is a drive letter in the path).

6.5.2 `type`

```
[type]: pkg|dir|git
```

The repository type. The type must be omitted for the base repository. If the type is omitted for a prerequisite/complement repository, then it is guessed from its `location` value as described in **`bpkg-rep-add(1)`**.

6.5.3 `role`

```
[role]: base|prerequisite|complement
```

The repository role. The `role` value can be omitted for the base repository only.

6.5.4 `trust`

```
[trust]: <fingerprint>
```

The repository fingerprint to trust. The `trust` value can only be specified for prerequisite and complement repositories and only for repository types that support authentication (currently only `pkg`). The *fingerprint* value should be an SHA256 repository fingerprint represented as 32 colon-separated hex digit pairs. The repository in question is only trusted for use as a prerequisite or complement of this repository. If it is also used by other repositories or is added to the configuration by the user, then such uses cases are authenticated independently.

6.5.5 `url`

```
[url]: <url>
```

The repository's web interface (`brep`) URL. It can only be specified for the base repository (the web interface URLs for prerequisite/complement repositories can be extracted from their respective manifests).

For example, given the following `url` value:

```
url: https://example.org/hello/
```

The package details page for `libfoo` located in this repository will be `https://example.org/hello/libfoo`.

The web interface URL can also be specified as relative to the repository location (the `location` value). In this case *url* should start with two path components each being either `.` or `..`. If the first component is `..`, then the `www`, `pkg` or `bpkg` domain component, if any, is removed from the `location` URL host, just like when deriving the repository name.

Similarly, if the second component is `..`, then the `pkg` or `bpkg` path component, if any, is removed from the `location` URL path, again, just like when deriving the repository name.

Finally, the version component is removed from the `location` URL path, the rest (after the two `./..` components) of the `url` value is appended to it, and the resulting path is normalized with all remaining `..` and `.` applied normally.

For example, assuming repository location is:

```
https://pkg.example.org/test/pkg/1/hello/stable
```

The following listing shows some of the possible combinations (the <> marker is used to highlight the changes):

```
./.          ->  https://pkg.example.org/test/pkg/hello/stable
../.         ->  https://<  >example.org/test/pkg/hello/stable
./..         ->  https://pkg.example.org/test/<  >hello/stable
../..        ->  https://<  >example.org/test/<  >hello/stable
././..       ->  https://pkg.example.org/test/pkg/hello<     >
../../../..  ->  https://<  >example.org/test<               >
```

The rationale for the relative web interface URLs is to allow deployment of the same repository to slightly different configuration, for example, during development, testing, and public use. For instance, for development we may use the `https://example.org/pkg/` setup while in production it becomes `https://pkg.example.org/`. By specifying the web interface location as, say, `../.`, we can run the web interface at these respective locations using a single repository manifest.

6.5.6 `email`

```
[email]: <email> [; <comment>]
```

The repository email address. It must and can only be specified for the base repository. The email address is displayed by the web interface (`brep`) in the repository about page and could be used to contact the maintainers about issues with the repository.

6.5.7 `summary`

```
[summary]: <text>
```

The short description of the repository. It must and can only be specified for the base repository.

6.5.8 `description`

```
[description]: <text>
```

The detailed description of the repository. It can only be specified for the base repository.

In the web interface (`brep`) the description is formatted into one or more paragraphs using blank lines as paragraph separators. Specifically, it is not represented as `<pre>` so any kind of additional plain text formatting (for example, lists) will be lost and should not be used in the description.

6.5.9 `certificate`

```
[certificate]: <pem>
```

The X.509 certificate for the repository. It should be in the PEM format and can only be specified for the base repository. Currently only used for the **`pkg`** repository type.

The certificate should contain the `CN` and `O` components in the subject as well as the `email:` component in the subject alternative names. The `CN` component should start with `name:` and continue with the repository name prefix/wildcard (without trailing slash) that will be used to verify the repository name(s) that are authenticated with this certificate. See **`bpkg-repository-signing(1)`** for details.

If this value is present then the `packages.manifest` file must be signed with the corresponding private key and the signature saved in the `signature.manifest` file. See Signature Manifest for details.

6.5.10 `fragment`

```
[fragment]: <string>
```

The repository fragment id this repository belongs to.

6.6 Repository List Manifest

@@ TODO See the Repository Chaining document for more information on the terminology and semantics.

The repository list manifest (the `repositories.manifest` file found in the repository root directory) describes the repository. It starts with an optional header manifest optionally followed by a sequence of repository manifests consisting of the base repository manifest (that is, the manifest for the repository that is being described) as well as manifests for its prerequisite and complement repositories. The individual repository manifests can appear in any order and the base repository manifest can be omitted.

The `fragment` values can only be present in a merged `repositories.manifest` file for a multi-fragment repository.

As an example, a repository manifest list for the `math/testing` repository could look like this:

```
# math/testing
#
: 1
min-bpkg-version: 0.14.0
:
email: math-pkg@example.org
summary: Math package repository
:
role: complement
```

```
location: ../stable
:
role: prerequiste
location: https://pkg.example.org/1/misc/testing
```

Here the first manifest describes the base repository itself, the second manifest – a complement repository, and the third manifest – a prerequisite repository. Note that the complement repository's location is specified as a relative path. For example, if the base repository location were:

```
https://pkg.example.org/1/math/testing
```

Then the completement's location would be:

```
https://pkg.example.org/1/math/stable
```

The header manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
[min-bpkg-version]: <ver>
[compression]: <compressions>
```

6.6.1 `min-bpkg-version`

```
[min-bpkg-version]: <ver>
```

The earliest version of **`bpkg`** that is compatible with this repository. Note that if specified, it must be the first value in the header.

6.6.2 `compression`

```
[compression]: <compressions>

<compressions> = <compression> [ <compression>]*
```

Available compressed variants of the `packages.manifest` file. The format is a space-separated list of the compression methods. The `none` method means no compression. Absent `compression` value is equivalent to specifying it with the `none` value.

6.7 Signature Manifest for `pkg` Repositories

The signature manifest (the `signature.manifest` file found in the **`pkg`** repository root directory) contains the signature of the repository's `packages.manifest` file. In order to detect the situation where the downloaded `signature.manifest` and `packages.manifest` files belong to different updates, the manifest contains both the checksum and the signature (which is the encrypted checksum). We cannot rely on just the signature since a mismatch could mean either a split update or tampering. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
sha256sum: <sum>
signature: <sig>
```

6.7.1 `sha256sum`

```
sha256sum: <sum>
```

The SHA256 checksum of the `packages.manifest` file. The *sum* value should be 64 characters long (that is, just the SHA256 value, no file name or any other markers), be calculated in the binary mode, and use lower-case letters.

6.7.2 `signature`

```
signature: <sig>
```

The signature of the `packages.manifest` file. It should be calculated by encrypting the above `sha256sum` value with the repository certificate's private key and then `base64`-encoding the result.