The `build2` Build Bot

Copyright © 2014-2024 the build2 authors (see the AUTHORS file).
Permission is granted to copy, distribute and/or modify this document under the terms of the MIT License.

Revision `0.18`, June 2024
This revision of the document describes the `build2` build bot `0.18.x` series.

Table of Contents

Preface 1
1 Introduction 1
2 Architecture 1
2.1 Configurations 2
2.1.1 Build Machine Configuration 2
2.1.2 Build Target Configuration 3
2.1.3 Build Package Configuration 4
2.1.4 Auxiliary Machines and Configurations 4
2.2 Machine Header Manifest 5
2.2.1 `id` 5
2.2.2 `name` 5
2.2.3 `summary` 6
2.2.4 `role` 6
2.2.5 `ram-minimum`, `ram-maximum` 6
2.3 Machine Manifest 6
2.3.1 `type` 6
2.3.2 `mac` 6
2.3.3 `options` 7
2.3.4 `changes` 7
2.4 Task Manifest 7
2.4.1 `name` 8
2.4.2 `version` 8
2.4.3 `repository-url` 8
2.4.4 `repository-type` 8
2.4.5 `trust` 9
2.4.6 `requires, tests, examples, benchmarks` 9
2.4.7 `dependency-checksum` 9
2.4.8 `machine` 9
2.4.9 `auxiliary-machine` 9
2.4.10 `target` 10
2.4.11 `environment` 10
2.4.12 `auxiliary-environment` 10
2.4.13 `target-config` 11
2.4.14 `package-config` 11
2.4.15 `host` 11
2.4.16 `warning-regex` 11
2.4.17 `interactive` 12
2.4.18 `worker-checksum` 12
2.5 Result Manifest 12
2.5.1 `name` 13
2.5.2 `version` 13
2.5.3 `status` 13
2.5.4 `*-status` 14
2.5.5 `*-log` 14

2.5.6 dependency-checksum 14
2.5.7 worker-checksum 14
2.6 Task Request Manifest 14
2.6.1 agent 15
2.6.2 toolchain-name 15
2.6.3 toolchain-version 15
2.6.4 interactive-mode 15
2.6.5 interactive-login 15
2.6.6 fingerprint 16
2.6.7 auxiliary-ram 16
2.7 Task Response Manifest 16
2.7.1 session 16
2.7.2 challenge 16
2.7.3 result-url 17
2.7.4 *-upload-url 17
2.7.5 agent-checksum 17
2.8 Result Request Manifest 17
2.8.1 session 17
2.8.2 challenge 17
2.8.3 agent-checksum 18
2.9 Worker Logic 18
2.9.1 Bindist Result Manifest 37
2.10 Controller Logic 37

Preface

This document describes `bbot`, the `build2` build bot. For the build bot command line interface refer to the **`bbot-agent(1)`** and **`bbot-worker(1)`** man pages.

1 Introduction

2 Architecture

The `bbot` architecture includes several layers for security and manageability. At the top we have a `bbot` running in the *controller* mode. The controller monitors various *build sources* for *build tasks*. For example, a controller may poll a `brep` instances for any new packages to built as well as monitor a **`git`** repository for any new commits to test. There can be several layers of controllers with `brep` being just a special kind. A machine running a `bbot` instance in the controller mode is called a *controller host*.

Below the controllers we have a `bbot` running in the *agent* mode normally on Build OS. The agent polls its controllers for *build tasks* to perform. A machine running a `bbot` instance in the agent mode is called a *build host*.

The actual building is performed in the virtual machines and/or containers that are executed on the build host. Inside virtual machines/containers, `bbot` is running in the *worker mode* and receives build tasks from its agent. Virtual machines and containers running a `bbot` instance in the worker mode are collectively called *build machines*.

In addition to a build machine, a build task may also require one or more *auxiliary machines* which provide additional components that are required for building or testing a package and that are impossible or impractical to provide as part of the build machine itself.

Let's now examine the workflow in the other direction, that is, from a worker to a controller. Once a build machine (plus auxiliary machines, if any) are booted (by the agent), the worker inside the build machine connects to the TFTP server running on the build host and downloads the *build task manifest*. It then proceeds to perform the build task and uploads the *build artifacts archive*, if any, followed by the *build result manifest* (which includes build logs) to the TFTP server.

Unlike build machines, auxiliary machines are not expected to run `bbot`. Instead, on boot, they are expected to upload to the TFTP server a list of environment variables to propagate to the build machine (see the `auxiliary-environment` task manifest value as well as Worker Logic for details).

Once an agent receives a build task for a specific build machine, it goes through the following steps. First, it creates a directory on its TFTP server with the *machine name* as its name and places the build task manifest inside. Next, it makes a throw-away snapshot of the build machine and boots it. After booting the build machine, the agent monitors the machine directory on its TFTP server for the build result manifest (uploaded by the worker once the build

has completed). Once the result manifest is obtained, the agent shuts down the build machine and discards its snapshot.

To obtains a build task the agent polls via HTTP/HTTPS one or more controllers. Before each poll request the agent enumerates the available build machines and sends this information as part of the request. The controller responds with a build task manifest that identifies a specific build machine to use.

In the task request the agent specifies if only non-interactive, interactive, or both build kinds are supported. If interactive builds are supported, it additionally provides the login information for interactive build sessions. If the controller responds with an interactive build task, then its manifest specifies the breakpoint the worker must stop the task execution at and prompt the user whether to continue or abort the execution. The user can log into the build machine, potentially perform some troubleshooting, and, when done, either answer the prompt or just shutdown the machine.

If the controller has higher-level controllers (for example, `brep`), then it aggregates the available build machines from its agents and polls these controllers (just as an agent would), forwarding build tasks to suitable agents. In this case we say that the *controller act as an agent*. The controller may also be configured to monitor build sources, such as SCM repositories, directly in which case it generates build tasks itself.

In this architecture the build results and optional build artifacts are propagated up the chain: from a worker, to its agent, to its controller, and so on. A controller that is the final destination of a build result uses email to notify interested parties of the outcome. For example, `brep` would send a notification to the package owner if the build failed. Similarly, a `bbot` controller that monitors a **`git`** repository would send an email to a committer if their commit caused a build failure. The email would include a link (normally HTTP/HTTPS) to the build logs hosted by the controller. The build artifacts, such as generated binary distribution packages, are normally made available for the interested parties to download. See Build Artifacts Upload for details on the `brep` controller's implementation of the build artifacts upload handling.

2.1 Configurations

The `bbot` architecture distinguishes between a *build machine configuration*, *build target configuration*, and a *build package configuration*. The machine configuration captures the operating system, installed compiler toolchain, and so on. The same build machine may be used to "generate" multiple *build target configurations*. For example, the same machine can normally be used to produce debug/optimized builds.

2.1.1 Build Machine Configuration

The machine configuration is *approximately* encoded in its *machine name*. The machine name is a list of components separated with −. Components cannot be empty and must contain only alpha-numeric characters, underscores, dots, and pluses with the whole id being a portably-valid path component.

The encoding is approximate in a sense that it captures only what's important to distinguish in a particular `bbot` deployment.

The first three components normally identify the architecture, operating system, and optional variant. They have the following recommended form:

```
<arch>-[<class>_]<os>[_<version>][-<variant>]
```

For example:

```
x86_64-windows
x86_64-windows_10
x86_64-windows_10.1607
x86_64-windows_10-devmode
x86_64-bsd_freebsd_10
x86_64-linux_ubuntu_16.04
x86_64-linux_rhel_9.2-bindist
aarch64-macos_10.12
```

The last component normally identifies the installed compiler toolchain and has the following recommended form:

```
<id>[_<version>][_<vendor>][_<runtime>]
```

For example:

```
gcc
gcc_6
gcc_6.3
gcc_6.3_mingw_w64
clang_3.9
clang_3.9_libc++
msvc_14
msvc_14.3
clang_15.0_msvc_msvc_17.6
clang_16.0_llvm_msvc_17.6
```

Some examples of complete machine names:

```
x86_64-windows_10-msvc_14.3
x86_64-macos_10.12-clang_10.0
aarch64-linux_ubuntu_16.04-gcc_6.3
aarch64-linux_rhel_9.2-bindist-gcc_11
```

2.1.2 Build Target Configuration

Similarly, the build target configuration is encoded in a *configuration name* using the same overall format. As described in Controller Logic, target configurations are generated from machine configurations. As a result, it usually makes sense to have the first component identify the operating systems and the second component – the compiler toolchain with the rest identifying a particular target configuration variant, for example, optimized, sanitized, etc:

```
[<class>_]<os>[_<version>]-<toolchain>[-<variant>]
```

For example:

```
windows_10-msvc_17.6
windows_10-msvc_17.6-O2
windows_10-msvc_17.6-static_O2
windows_10-msvc_17.6-relocatable
windows_10-clang_16.0_llvm_msvc_17.6_lld
linux_debian_12-clang_16_libc++-static_O3
```

Note that there is no `<arch>` component in a build target configuration: this information is best conveyed as part of `<target>` as described in Controller Logic.

2.1.3 Build Package Configuration

A package can be built in multiple package configurations per target configuration. A build package configuration normally specifies the options and/or the package configuration variables that need to be used for the build. It may also include the information regarding the dependency packages which need to additionally be configured. The build package configurations originate from the package manifest `*-build-config`, `*-builds`, `*-build-include`, and `*-build-exclude` values. See Package Manifest for more information on these values.

2.1.4 Auxiliary Machines and Configurations

Besides the build machine and the build configuration that is derived from it, a package build may also involve one or more *auxiliary machines* and the corresponding *auxiliary configurations*.

An auxiliary machine provides additional components that are required for building or testing a package and that are impossible or impractical to provide as part of the build machine itself. For example, a package may need access to a suitably configured database, such as PostgreSQL, in order to run its tests.

The auxiliary machine name follows the same overall format as the build machine name except that the last component captures the information about the additional component in question rather than the compiler toolchain. For example:

```
x86_64-linux_debian_12-postgresql_16
aarch64-linux_debian_12-mysql_8
```

The auxiliary configuration name is automatically derived from the machine name by removing the `<arch>` component. For example:

```
linux_debian_12-postgresql_16
linux_debian_12-mysql_8
```

Note that there is no generation of multiple auxiliary configurations from the same auxiliary machine since that would require some communication of the desired configuration variant to the machine.

2.2 Machine Header Manifest

@@ TODO: need ref to general manifest overview in bpkg, or, better yet, move it to libbutl and ref to that from both places.

The build machine header manifest contains basic information about a build machine on the build host. A list of machine header manifests is sent by `bbot` agents to controllers. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
id: <machine-id>
name: <machine-name>
summary: <string>
[role]: build|auxiliary
[ram-minimum]: <kib>
[ram-maximum]: <kib>
```

For example:

```
id: x86_64-windows_10-msvc_14-1.3
name: x86_64-windows_10-msvc_14
summary: Windows 10 build 1607 with VC 14 update 3

id: aarch64-linux_debian_12-postgresql_16-1.0
name: aarch64-linux_debian_12-postgresql_16
summary: Debian 12 with PostgreSQL 16 test user/database
role: auxiliary
ram-minimum: 2097152
ram-maximum: 4194304
```

2.2.1 `id`

```
id: <machine-id>
```

The unique machine version/revision/build identifier. For virtual machines this can be the disk image checksum. For a container this can be UUID that is re-generated every time a container filesystem is altered.

Note that we assume that a different machine identifier is assigned on any change that may affect the build result.

2.2.2 `name`

```
name: <machine-name>
```

The machine name.

2.2.3 `summary`

```
summary: <string>
```

The one-line description of the machine.

2.2.4 `role`

```
[role]: build|auxiliary
```

The machine role. If unspecified, then `build` is assumed.

2.2.5 `ram-minimum`, `ram-maximum`

```
[ram-minimum]: <kib>
[ram-maximum]: <kib>
```

The minimum and the maximum amount of RAM in KiB that the machine requires. The maximum amount is interpreted as the amount beyond which there will be no benefit. If unspecified, then it is assumed the machine will run with any minimum amount a deployment will provide and will always benefit from more RAM, respectively. Neither value should be `0`.

2.3 Machine Manifest

The build machine manifest contains the complete description of a build machine on the build host (see the Build OS documentation for their origin and location). The machine manifest starts with the machine header manifest with all the header values appearing before any non-header values. The non-header part of manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
type: kvm|nspawn
[mac]: <addr>
[options]: <machine-options>
[changes]: <text>
```

2.3.1 `type`

```
type: kvm|nspawn
```

The machine type. Valid values are `kvm` (QEMU/KVM virtual machine) and `nspawn` (`systemd-nspawn` container).

2.3.2 `mac`

```
[mac]: <addr>
```

The fixed MAC address for the machine. Must be in the hexadecimal, comma-separated format. For example:

```
mac: de:ad:be:ef:de:ad
```

If it is not specified, then a random address is generated on the first machine bootstrap which is then reused for each build/re-bootstrap. Note that if you specify a fixed address, then the machine can only be used by a single `bbot` agent.

2.3.3 `options`

```
[options]: <machine-options>
```

The list of machine options. The exact semantics is machine type-dependent (see below). A single level of quotes (either single or double) is removed in each option before being passed on. Options can be separated with spaces or newlines.

For `kvm` machines, if this value is present, then it replaces the default network and disk configuration when starting the QEMU/KVM hypervisor. The options are pre-processed by replacing the question mark in `ifname=?` and `mac=?` strings with the network interface and MAC address, respectively.

2.3.4 `changes`

```
[changes]: <text>
```

The description of machine changes in this version.

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. For example:

```
changes: 1.1: initial version
changes: 1.2: increased disk size to 30GB
```

Or:

```
changes:
\
1.1
  - initial version

1.2
  - increased disk size to 30GB
  - upgraded bootstrap baseutils
\
```

2.4 Task Manifest

The task manifest describes a build task. It consists of two groups of values. The first group defines the package to build. The second group defines the build configuration to use for building the package. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
name: <package-name>
version: <package-version>
#location: <package-url>
repository-url: <repository-url>
[repository-type]: pkg|git|dir
[trust]: <repository-fp>
[requires]: <package-requirements>
[tests]: <dependency-package>
[examples]: <dependency-package>
[benchmarks]: <dependency-package>
[dependency-checksum]: <checksum>

machine: <machine-name>
[auxiliary-machine]: <machine-name>
[auxiliary-machine-<name>]: <machine-name>
target: <target-triplet>
[environment]: <environment-name>
[auxiliary-environment]: <environment-vars>
[target-config]: <tgt-config-args>
[package-config]: <pkg-config-args>
[host]: true|false
[warning-regex]: <warning-regex>
[interactive]: <breakpoint>
[worker-checksum]: <checksum>
```

2.4.1 `name`

```
name: <package-name>
```

The package name to build.

2.4.2 `version`

```
version: <package-version>
```

The package version to build.

2.4.3 `repository-url`

```
repository-url: <repository-url>
```

The URL of the repository that contains the package and its dependencies.

2.4.4 `repository-type`

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

The repository type (see `repository-url` for details). Alternatively, the repository type can be specified as part of the URL scheme. See **`bpkg-repository-types(1)`** for details.

2.4.5 `trust`

```
[trust]: <repository-fp>
```

The SHA256 repository certificate fingerprint to trust (see the `bpkg --trust` option for details). This value may be specified multiple times to establish the authenticity of multiple certificates. If the special `yes` value is specified, then all repositories will be trusted without authentication (see the `bpkg --trust-yes` option).

Note that while the controller may return a task with `trust` values, whether they will be used is up to the agent's configuration. For example, some agents may only trust their internally-specified fingerprints to prevent the "man in the middle" attacks.

2.4.6 `requires, tests, examples, benchmarks`

The primary package manifest values that need to be known by the `bbot` worker before it retrieves the primary package manifest. See Package Manifest for more information on these values.

The controller copies these values from the primary package manifest, except those `tests`, `examples`, and `benchmarks` values which should be excluded from building due to their `builds`, `build-include`, and `build-exclude` manifest values.

2.4.7 `dependency-checksum`

```
[dependency-checksum]: <checksum>
```

The package dependency checksum received as a part of the previous build task result (see Result Manifest).

2.4.8 `machine`

```
machine: <machine-name>
```

The name of the build machine to use.

2.4.9 `auxiliary-machine`

```
[auxiliary-machine]: <machine-name>
[auxiliary-machine-<name>]: <machine-name>
```

The names of the auxiliary machines to use. These values correspond to the `build-auxiliary` and `build-auxiliary-<name>` values in the package manifest. While there each value specifies an auxiliary configuration pattern, here it specifies the concrete auxiliary machine name that was picked by the controller from the list of available auxiliary machines (sent as part of the task request) that match this pattern.

2.4.10 `target`

```
target: <target-triplet>
```

The target to build for.

Compared to the autotools terminology, the `machine` value corresponds to `--build` (the machine we are building on) and `target` – to `--host` (the machine we are building for). While we use essentially the same *target triplet* format as autotools for `target`, it is not flexible enough for `machine`.

2.4.11 `environment`

```
[environment]: <environment-name>
```

The name of the build environment to use. See Worker Logic for details.

2.4.12 `auxiliary-environment`

```
[auxiliary-environment]: <environment-vars>
```

The environment variables describing the auxiliary machines. If any `auxiliary-machine*` values are specified, then after starting such machines, the agent prepares a combined list of environment variables that were uploaded by such machines and passes it in this value to the worker.

The format of this value is a list of environment variable assignments one per line, in the form:

```
<name>=<value>
```

Whitespaces before `<name>`, around =, and after `<value>` as well as blank lines and lines that start with # are ignored. The `<name>` part must only contain capital alphabetic, numeric, and _ characters. The `<value>` part as a whole can be single (’ ’) or double (" ") quoted. For example:

```
DATABASE_HOST=192.168.0.1
DATABASE_PORT=1245
DATABASE_USER=’John "Johnny" Doe’
DATABASE_NAME=" test database "
```

If the corresponding machine is specified as `auxiliary-machine-<name>`, then its environment variables are prefixed with capitalized `<name>_`. For example:

```
auxiliary-machine-pgsql: x86_64-linux_debian_12-postgresql_16
auxiliary-environment:
\
PGSQL_DATABASE_HOST=192.168.0.1
PGSQL_DATABASE_PORT=1245
...
\
```

2.4.13 `target-config`

```
[target-config]: <tgt-config-args>
```

The additional target configuration options and variables. A single level of quotes (either single or double) is removed in each value before being passed to `bpkg`. For example, the following value:

```
target-config: config.cc.coptions="-O3 -stdlib='libc++'"
```

Will be passed to `bpkg` as the following (single) argument:

```
config.cc.coptions=-O3 -stdlib='libc++'
```

Values can be separated with spaces or newlines. See Controller Logic for details.

2.4.14 `package-config`

```
[package-config]: <pkg-config-args>
```

The primary package manifest `*-build-config` value for the build configuration the build task is issued for. See Package Manifest for more information on this value. A single level of quotes (either single or double) is removed in each value before being passed to `bpkg`. For example, the following value:

```
package-config: "?libcurl ~7.76.0"
```

Will be passed to `bpkg` as the following (single) argument:

```
?libcurl ~7.76.0
```

Values can be separated with spaces or newlines. See Controller Logic for details.

2.4.15 `host`

```
[host]: true|false
```

If `true`, then the build target configuration is self-hosted. If not specified, `false` is assumed. See Controller Logic for details.

2.4.16 `warning-regex`

```
[warning-regex]: <warning-regex>
```

Additional regular expressions that should be used to detect warnings in the build logs. Note that only the first 512 bytes of each log line is considered.

A single level of quotes (either single or double) is removed in each expression before being used for search. For example, the following value:

```
warning-regex: "warning C4\d{3}: "
```

Will be treated as the following (single) regular expression (with a trailing space):

```
warning C4\d{3}:
```

Expressions can be separated with spaces or newlines. They will be added to the following default list of regular expressions that detect the `build2` toolchain warnings:

```
^warning:
^.+: warning:
```

Note that this built-in list also covers GCC and Clang warnings (for the English locale).

2.4.17 `interactive`

```
[interactive]: <breakpoint>
```

The task execution step to stop at. Can only be present if the agent has specified `interactive-mode` with either the `true` or `both` value in the task request.

The breakpoint can either be a primary step id of the worker script or the special `error` or `warning` value. There is also the special `none` value which never interrupts the task execution. See Worker Logic for details.

2.4.18 `worker-checksum`

```
[worker-checksum]: <checksum>
```

The worker checksum received as a part of the previous build task result (see Result Manifest).

2.5 Result Manifest

The result manifest describes a build result. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
name: <package-name>
version: <package-version>

status:                  <status>
[configure-status]:      <status>
[update-status]:         <status>
[test-status]:           <status>
[install-status]:        <status>
[bindist-status]:        <status>
[sys-install-status]:    <status>
[test-installed-status]: <status>
[sys-uninstall-status]:  <status>
[uninstall-status]:      <status>
[upload-status]:         <status>

[configure-log]:      <text>
[update-log]:         <text>
```

```
[test-log]:           <text>
[install-log]:        <text>
[bindist-log]:        <text>
[sys-install-log]:    <text>
[test-installed-log]: <text>
[sys-uninstall-log]:  <text>
[uninstall-log]:      <text>
[upload-log]:         <text>

[worker-checksum]:     <checksum>
[dependency-checksum]: <checksum>
```

2.5.1 `name`

```
name: <package-name>
```

The package name from the task manifest.

2.5.2 `version`

```
version: <package-version>
```

The package version from the task manifest.

2.5.3 `status`

```
status: <status>
```

The overall (cumulative) build result status. Valid values are:

```
skip       # Package update and subsequent operations were skipped.
success    # All operations completed successfully.
warning    # One or more operations completed with warnings.
error      # One or more operations completed with errors.
abort      # One or more operations were aborted.
abnormal   # One or more operations terminated abnormally.
interrupt  # Task execution has been interrupted.
```

The `abort` status indicates that the operation has been aborted by `bbot`, for example, because it was consuming too many resources and/or was taking too long. Note that a task can be aborted both by the `bbot` worker as well as the agent. In the later case the whole machine is shut down and no operation-specific status or logs will be included (@@ Maybe we should just include 'log:' with commands that start VM, for completeness?).

The `abnormal` status indicates that the operation has terminated abnormally, for example, due to the package manager or build system crash.

The `interrupt` status indicates that the task execution has been interrupted, for example, to reassign resources to a higher priority task.

Note that the overall `status` value should appear before any per-operation `*-status` values.

The `skip` status indicates that the received from the controller build task checksums have not changed and the task execution has therefore been skipped under the assumption that it would have produced the same result. See `agent-checksum`, `worker-checksum`, and `dependency-checksum` for details.

2.5.4 `*-status`

```
[*-status]: <status>
```

The per-operation result status. Note that the `*-status` values should appear in the same order as the corresponding operations were performed and for each `*-status` there should be the corresponding `*-log` value. Currently supported operation names:

```
configure
update
test
install
bindist
sys-install
test-installed
sys-uninstall
uninstall
upload
```

2.5.5 `*-log`

```
[*-log]: <text>
```

The per-operation result log. Note that the `*-log` values should appear last and in the same order as the corresponding `*-status` values. For the list of supported operation names refer to the `*-status` value description.

2.5.6 `dependency-checksum`

```
[dependency-checksum]: <checksum>
```

The package dependency checksum obtained as a byproduct of the package configuration operation. See **`bpkg-pkg-build(1)`** command's `--rebuild-checksum` option for details.

2.5.7 `worker-checksum`

```
[worker-checksum]: <checksum>
```

The version of the worker logic used to perform the package build task.

2.6 Task Request Manifest

An agent (or controller acting as an agent) sends a task request to its controller via HTTP/HTTPS POST method (@@ URL/API endpoint). The task request starts with the task request manifest followed by a list of machine header manifests. The task request manifest

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

The controller is expected to pick each offered machine header manifest only once. If an agent is capable of running multiple instances of the same machine, then it must send the matching number of machine header manifests for such a machine.

```
agent: <name>
toolchain-name: <name>
toolchain-version: <standard-version>
[interactive-mode]: false|true|both
[interactive-login]: <login>
[fingerprint]: <agent-fingerprint>
[auxiliary-ram]: <kib>
```

2.6.1 `agent`

```
agent: <name>
```

The name of the agent host (`hostname`). The name should be unique in a particular `bbot` deployment.

2.6.2 `toolchain-name`

```
toolchain-name: <name>
```

The `build2` toolchain name being used by the agent.

2.6.3 `toolchain-version`

```
toolchain-version: <standard-version>
```

The `build2` toolchain version being used by the agent.

2.6.4 `interactive-mode`

```
[interactive-mode]: false|true|both
```

The agent's capability to perform build tasks only non-interactively (`false`), only interactively (`true`), or both (`both`).

If it is not specified, then the `false` value is assumed.

2.6.5 `interactive-login`

```
[interactive-login]: <login>
```

The login information for the interactive build session. Must be present only if `interactive-mode` is specified with the `true` or `both` value.

2.6.6 `fingerprint`

```
[fingerprint]: <agent-fingerprint>
```

The SHA256 fingerprint of the agent's public key. An agent may be configured not to use the public key-based authentication in which case it does not include this value. However, the controller may be configured to require the authentication in which case it should respond with the 401 (unauthorized) HTTP status code.

2.6.7 `auxiliary-ram`

```
[auxiliary-ram]: <kib>
```

The amount of RAM in KiB that is available for running auxiliary machines. If unspecified, then assume there is no hard limit (that is, the agent can allocate up to the host's available RAM minus the amount required to run the build machine).

2.7 Task Response Manifest

A controller sends the task response manifest in response to the task request initiated by an agent. The response is delivered as a result of the POST method. The task response starts with the task response manifest optionally followed by the task manifest. The task response manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
session: <id>
[challenge]: <text>
[result-url]: <url>
[*-upload-url]: <url>
[agent-checksum]: <checksum>
```

2.7.1 `session`

```
session: <id>
```

The identifier assigned to this session by the controller. An empty value indicates that the controller has no tasks at this time in which case all the following values as well as the task manifest are absent.

2.7.2 `challenge`

```
[challenge]: <text>
```

The random, 64 characters long string (nonce) used to challenge the agent's private key. If present, then the agent must sign this string and include the signature in the result request (see below).

The signature should be calculated by encrypting the string with the agent's private key and then `base64`-encoding the result.

2.7.3 `result-url`

```
[result-url]: <url>
```

The URL to POST (upload) the result request to.

2.7.4 `*-upload-url`

```
[*-upload-url]: <url>
```

The URLs to upload the build artifacts to, if any, via the HTTP `POST` method using the `multipart/form-data` content type (see Build Artifacts Upload for details on the upload protocol). The substring matched by `*` in `*-upload-url` denotes the upload type.

2.7.5 `agent-checksum`

```
[agent-checksum]: <checksum>
```

The agent checksum received as a part of the previous build task result request (see Result Request Manifest).

2.8 Result Request Manifest

On completion of a task an agent (or controller acting as an agent) sends the result (upload) request to the controller via the POST method using the URL returned in the task response (see above). The result request starts with the result request manifest followed by the result manifest. Note that there is no result response and only a successful but empty POST result is returned. The result request manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
session: <id>
[challenge]: <text>
[agent-checksum]: <checksum>
```

2.8.1 `session`

```
session: <session-id>
```

The session id as returned by the controller in the task response.

2.8.2 `challenge`

```
[challenge]: <text>
```

The answer to the private key challenge as posed by the controller in the task response. It must be present only if the `challenge` value was present in the task response.

2.8.3 `agent-checksum`

```
[agent-checksum]: <checksum>
```

The version of the agent logic used to perform the package build task.

2.9 Worker Logic

The `bbot` worker builds each package in a *build environment* that is established for a particular build target. The environment has three components: the execution environment (environment variables, etc), build system modules, as well as configuration options and variables.

Setting up of the execution environment is performed by an executable (script, batch file, etc). Specifically, upon receiving a build task, if it specifies the environment name then the worker looks for the environment setup executable with this name in a specific directory and for the executable called `default` otherwise. Not being able to locate the environment executable is an error.

In addition to the environment executable, if the task requires any auxiliary machines, then the `auxiliary-environment` value from the task manifest is incorporated into the execution environment.

Specifically, once the environment setup executable is determined, the worker re-executes itself in the auxiliary environment and as that executable passing to it as command line arguments the target name, the path to the `bbot` worker to be executed once the environment is setup, and any additional options that need to be propagated to the re-executed worker. The environment setup executable is executed in the build directory as its current working directory. The build directory contains the build task `task.manifest` file.

The environment setup executable sets up the necessary execution environment for example by adjusting `PATH` or running a suitable `vcvars` batch file. It then re-executes itself as the `bbot` worker passing to it as command line arguments (in addition to worker options) the list of build system modules (`<env-modules>`) and the list of configuration options and variables (`<env-config-args>`). The environment setup executable must execute the `bbot` worker in the build directory as the current working directory.

The re-executed `bbot` worker then proceeds to test the package from the repository by executing the following commands, collectively called a *worker script*. Each command has a unique *step id* that can be used as a breakpoint and normally as a prefix in the `<tgt-config-args>`, `<env-config-args>`, and `<env-modules>` values as discussed in Controller Logic as well as in the `<pkg-config-args>` values (see below). The <>-values are from the task manifest and the environment though some are assigned by the worker during the script execution (configuration directories, UUIDs, etc). In particular, the `<pkg-config-args>` (prefixed global options and variables), `<pkg-config-opts>` (unprefixed options), `<pkg-config-vars>` (unprefixed variables), `<dependency-name>`, `<dependency-version-constraint>`, and `<dep-config-vars>` values result from parsing the `package-config` task manifest

value.

Some prefix step ids have fallback step ids which are used in the absence of the primary step id values. If the prefix step id differs from the breakpoint step id and/or has the fallback step ids, then they are listed in parenthesis: the prefix id before the colon and the fallback ids after it.

Some commands have no target configuration or environment options or variables. Such commands have only breakpoint step ids associated, which are listed in square brackets.

Note that the worker script varies for different primary package types. The `bbot` worker classifies the primary package based on the configuration type in which it is built: `module` (build system module packages), `host` (packages such as source code generators, marked with the `requires: host` manifest value; see Package Manifest for details), and `target` (all other packages).

Note also that the `*.configure.build` step configures potentially multiple packages (primary package, tests, etc) in potentially multiple configurations by always using the `bpkg.global.configure.build` prefix step id for global (as opposed to package-specific) **bpkg-pkg-build(1)** options. The `bpkg.global.configure.build` prefix id has no fallback ids.

Note finally that if no configuration variables are specified in the main package configuration, then the worker adds the `config.<name>.develop=false` configuration variable for the main package at the `bpkg.configure.build` step to trigger its package skeleton creation and loading. It also adds this variable for external test packages at this step and for the same purpose. This makes sure that these packages can be used as dependencies of dependents with configuration clauses. To keep the below listings concise, these variables are not shown.

Worker script for `target` packages:

```
# bpkg.create (bpkg.target.create : b.create, bpkg.create)
#
bpkg -V create <env-modules> \
     <env-config-args> <tgt-config-args> <pkg-config-args>

# bpkg.configure.add
#
bpkg -v add <repository-url>

# bpkg.configure.fetch
#
bpkg -v fetch --trust <repository-fp>

# bpkg.configure.build (
#   bpkg.global.configure.build,
#   (bpkg.target.configure.build : b.configure, bpkg.configure.build))
#
bpkg -v build --configure-only \
     <env-config-args> <tgt-config-args> <pkg-config-args> \
     [<pkg-config-opts>] \
     [{ <pkg-config-vars> }+] <package-name>/<package-version> \
```

```
     [([{ <test-config-vars> }+] \
       <test-package-name>[ <test-version-constraint>])...] \
     [([{ <dep-config-vars> }+] \
       (?|sys:)<dependency-name> \
       [<dependency-version-constraint>])...] \
     [?sys:<dependency-name>[ <dependency-version-constraint>]...]

# bpkg.update
#
bpkg -v update <package-name>

# If the test operation is supported by the package:
#
{
  # bpkg.test
  #
  bpkg -v test <package-name>
}

# For each (runtime) tests, examples, or benchmarks package referred
# to by the task manifest:
#
{
  # bpkg.test-separate.update ( : bpkg.update)
  #
  bpkg -v update <package-name>

  # bpkg.test-separate.test ( : bpkg.test)
  #
  bpkg -v test <package-name>
}

# If the install operation is supported by the package,
# config.install.root is specified, and no
# bpkg.bindist.{debian,fedora,archive} step is enabled:
#
{
  # bpkg.install
  #
  bpkg -v install <package-name>

  # If bbot.install.ldconfig step is enabled:
  #
  {
    # bbot.install.ldconfig
    #
    sudo ldconfig
  }
}

# If the install operation is supported by the package and
# bpkg.bindist.{debian,fedora,archive} step is enabled:
#
{
  # bpkg.bindist.{debian,fedora,archive}
  #
  bpkg -v bindist --distribution <distribution> \
       <env-config-args> <tgt-config-args> <pkg-config-args> \
       <package-name>
}

# If the install operation is supported by the package and
```

```
# bbot.sys-install step is enabled:
#
{
  # If <distribution> is 'debian':
  #
  {
    # bbot.sys-install.apt-get.update
    #
    sudo apt-get update

    # bbot.sys-install.apt-get.install
    #
    sudo apt-get install <distribution-package-file>...
  }
  #
  # Otherwise, if <distribution> is 'fedora':
  #
  {
    # bbot.sys-install.dnf.install
    #
    sudo dnf install <distribution-package-file>...
  }
  #
  # Otherwise, if <distribution> is 'archive':
  #
  {
    # For each package file:
    #
    {
      # bbot.sys-install.tar.extract
      #
      [sudo] tar -xf <distribution-package-file> \
                 <env-config-args> <tgt-config-args> <pkg-config-args>
    }

    # If bbot.sys-install.ldconfig step is enabled:
    #
    {
      # bbot.sys-install.ldconfig
      #
      sudo ldconfig
    }
  }
}

# If the main package is installed either from source or from the
# binary distribution package:
#
{
  # If the package contains subprojects that support the test
  # operation:
  #
  {
    # b.test-installed.create ( : b.create)
    #
    b -V create <env-modules> \
      <env-config-args> <tgt-config-args> <pkg-config-args>

    # For each test subproject:
    #
    {
      # b.test-installed.configure ( : b.configure)
```

```
      #
      b -v configure [<pkg-config-vars>]
    }

    # b.test-installed.test
    #
    b -v test
  }

  # If task manifest refers to any (runtime) tests, examples, or
  # benchmarks packages:
  #
  {
    # bpkg.test-separate-installed.create (
    #   bpkg.test-separate-installed.create_for_target :
    #     bpkg.test-separate-installed.create)
    #
    bpkg -V create <env-modules> \
         <env-config-args> <tgt-config-args> <pkg-config-args>

    # bpkg.test-separate-installed.configure.add (
    #  : bpkg.configure.add)
    #
    bpkg -v add <repository-url>

    # bpkg.test-separate-installed.configure.fetch (
    #   : bpkg.configure.fetch)
    #
    bpkg -v fetch --trust <repository-fp>

    # bpkg.test-separate-installed.configure.build (
    #   bpkg.global.configure.build,
    #   (bpkg.test-separate-installed.configure.build_for_target :
    #      bpkg.test-separate-installed.configure.build))
    #
    bpkg -v build --configure-only \
         <env-config-args> <tgt-config-args> <pkg-config-args> \
         ([{ <test-config-vars> }+] \
          <test-package-name>[ <test-version-constraint>])... \
         ?sys:<package-name>/<package-version> \
         [?sys:<dependency-name>[ <dependency-version-constraint>]...]

    # For each (runtime) tests, examples, or benchmarks package
    # referred to by the task manifest:
    #
    {
      # bpkg.test-separate-installed.update ( : bpkg.update)
      #
      bpkg -v update <package-name>

      # bpkg.test-separate-installed.test ( : bpkg.test)
      #
      bpkg -v test <package-name>
    }
  }
}

# If the main package is installed from the binary distribution package:
#
{
  # If <distribution> is 'debian':
  #
```

```
  {
    # bbot.sys-uninstall.apt-get.remove
    #
    sudo apt-get remove <distribution-package-name>...
  }
  #
  # Otherwise, if <distribution> is 'fedora':
  #
  {
    # bbot.sys-uninstall.dnf.remove
    #
    sudo dnf remove <distribution-package-name>...
  }
  #
  # Otherwise, if <distribution> is 'archive':
  #
  {
    # Noop.
  }
}

# If the main package is installed from source:
#
{
  # bpkg.uninstall
  #
  bpkg -v uninstall <package-name>
}

# If the install operation is supported by the package and
# bbot.bindist.upload step is enabled:
#
{
  # Move the generated binary distribution files to the
  # upload/bindist/<distribution>/ directory.
}

# If bbot.upload step is enabled and upload/ directory is not empty:
#
{
  # bbot.upload.tar.create
  #
  tar -cf upload.tar upload/

  # bbot.upload.tar.list
  #
  tar -tf upload.tar upload/
}

# end
#
# This step id can only be used as a breakpoint.
```

Worker script for `host` packages:

```
# If configuration is self-hosted:
#
{
  # bpkg.create (bpkg.host.create : b.create, bpkg.create)
  #
  bpkg -V create --type host -d <host-conf> <env-modules> \
```

```
       <env-config-args> <tgt-config-args> <pkg-config-args>
}
#
# Otherwise:
#
{
  # [bpkg.create]
  #
  b -V create(<host-conf>, cc) config.config.load=~host-no-warnings
  bpkg -v create --existing --type host -d <host-conf>
}

# bpkg.configure.add
#
bpkg -v add -d <host-conf> <repository-url>

# bpkg.configure.fetch
#
bpkg -v fetch -d <host-conf> --trust <repository-fp>

# If configuration is self-hosted and config.install.root is specified:
#
{
  # bpkg.create (bpkg.target.create : b.create, bpkg.create)
  #
  bpkg -V create -d <install-conf> <env-modules> \
       <env-config-args> <tgt-config-args> <pkg-config-args>

  # [bpkg.link]
  #
  bpkg -v link -d <install-conf> <host-conf>

  # bpkg.configure.add
  #
  bpkg -v add -d <install-conf> <repository-url>

  # bpkg.configure.fetch
  #
  bpkg -v fetch -d <install-conf> --trust <repository-fp>
}

# If task manifest refers to any build-time tests, examples, or
# benchmarks packages:
#
{
  # bpkg.create (bpkg.target.create : b.create, bpkg.create)
  #
  bpkg -V create -d <target-conf> <env-modules> \
       <env-config-args> <tgt-config-args> <pkg-config-args>

  # [bpkg.create]
  #
  b -V create(<module-conf>, cc) config.config.load=~build2-no-warnings
  bpkg -v create --existing --type build2 -d <module-conf>

  # [bpkg.link]
  #
  bpkg -v link -d <target-conf> <host-conf>
  bpkg -v link -d <target-conf> <module-conf>
  bpkg -v link -d <host-conf>   <module-conf>

  # If configuration is self-hosted and config.install.root is
```

```
  # specified:
  #
  {
    # [bpkg.link]
    #
    bpkg -v link -d <install-conf> <module-conf>
  }

  # bpkg.configure.add
  #
  bpkg -v add -d <target-conf> <repository-url>

  # bpkg.configure.fetch
  #
  bpkg -v fetch -d <target-conf> --trust <repository-fp>
}

# bpkg.configure.build (bpkg.global.configure.build)
#
# Notes:
#
# - Some parts may be omitted.
#
# - Parts related to different configurations have different prefix
#   step ids:
#
#   bpkg.host.configure.build   for <host-uuid>
#   bpkg.target.configure.build for <install-uuid>
#   bpkg.target.configure.build for <target-uuid>
#
# - All parts have the same fallback step ids: b.configure and
#   bpkg.configure.build.
#
bpkg -v build --configure-only \
<env-config-args> <tgt-config-args> <pkg-config-args> \
[<pkg-config-opts>] \
\
{ --config-uuid <host-uuid> \
  <env-config-args> <tgt-config-args> <pkg-config-args> \
  [<pkg-config-vars>] }+ \
<package-name>/<package-version> \
\
{ --config-uuid <install-uuid> \
  <env-config-args> <tgt-config-args> <pkg-config-args> \
  [<pkg-config-vars>] }+ \
<package-name>/<package-version> \
\
({ --config-uuid <host-uuid> \
   <env-config-args> <tgt-config-args> <pkg-config-args> \
   [<test-config-vars>] }+ \
 <runtime-test-package-name>[ <test-version-constraint>])... \
\
({ --config-uuid <target-uuid> \
   <env-config-args> <tgt-config-args> <pkg-config-args> \
   [<test-config-vars>] }+ \
 <buildtime-test-package-name>[ <test-version-constraint>])... \
\
({ --config-uuid <host-uuid> [--config-uuid <install-uuid>] \
   [<dep-config-vars>] }+ \
 (?|sys:)<dependency-name>[ <dependency-version-constraint>])... \
\
[?sys:<dependency-name>[ <dependency-version-constraint>]...]
```

```
# bpkg.update
#
bpkg -v update -d <host-conf> <package-name>

# If the test operation is supported by the package:
#
{
  # bpkg.test
  #
  bpkg -v test -d <host-conf> <package-name>
}

# If configuration is self-hosted, then for each runtime tests,
# examples, or benchmarks package referred to by the task manifest:
#
{
  # bpkg.test-separate.update ( : bpkg.update)
  #
  bpkg -v update -d <host-conf> <package-name>

  # bpkg.test-separate.test ( : bpkg.test)
  #
  bpkg -v test -d <host-conf> <package-name>
}

# For each build-time tests, examples, or benchmarks package referred
# to by the task manifest:
#
{
  # bpkg.test-separate.update ( : bpkg.update)
  #
  bpkg -v update -d <target-conf> <package-name>

  # bpkg.test-separate.test ( : bpkg.test)
  #
  bpkg -v test -d <target-conf> <package-name>
}

# If configuration is self-hosted, the install operation is supported
# by the package, config.install.root is specified, and no
# bpkg.bindist.{debian,fedora,archive} step is enabled:
#
{
  # bpkg.install
  #
  bpkg -v install -d <install-conf> <package-name>

  # If bbot.install.ldconfig step is enabled:
  #
  {
    # bbot.install.ldconfig
    #
    sudo ldconfig
  }
}

# If configuration is self-hosted, the install operation is supported
# by the package, and bpkg.bindist.{debian,fedora,archive} step is
# enabled:
#
{
```

```
  # bpkg.bindist.{debian,fedora,archive}
  #
  bpkg -v bindist --distribution <distribution> \
       <env-config-args> <tgt-config-args> <pkg-config-args> \
       <package-name>
}

# If the install operation is supported by the package and
# bbot.sys-install step is enabled:
#
{
  # If <distribution> is 'debian':
  #
  {
    # bbot.sys-install.apt-get.update
    #
    sudo apt-get update

    # bbot.sys-install.apt-get.install
    #
    sudo apt-get install <distribution-package-file>...
  }
  #
  # Otherwise, if <distribution> is 'fedora':
  #
  {
    # bbot.sys-install.dnf.install
    #
    sudo dnf install <distribution-package-file>...
  }
  #
  # Otherwise, if <distribution> is 'archive':
  #
  {
    # For each package file:
    #
    {
      # bbot.sys-install.tar.extract
      #
      [sudo] tar -xf <distribution-package-file> \
                 <env-config-args> <tgt-config-args> <pkg-config-args>
    }

    # If bbot.sys-install.ldconfig step is enabled:
    #
    {
      # bbot.sys-install.ldconfig
      #
      sudo ldconfig
    }
  }
}

# If the main package is installed either from source or from the
# binary distribution package:
#
{
  # If the package contains subprojects that support the test
  # operation:
  #
  {
    # b.test-installed.create ( : b.create)
```

```
    #
    b -V create <env-modules> \
      <env-config-args> <tgt-config-args> <pkg-config-args>

    # For each test subproject:
    #
    {
      # b.test-installed.configure ( : b.configure)
      #
      b -v configure [<pkg-config-vars>]
    }

    # b.test-installed.test
    #
    b -v test
  }

  # If task manifest refers to any tests, examples, or benchmarks
  # packages:
  #
  {
    # bpkg.test-separate-installed.create (
    #   bpkg.test-separate-installed.create_for_host :
    #     bpkg.test-separate-installed.create)
    #
    bpkg -V create --type host -d <host-conf> <env-modules> \
         <env-config-args> <tgt-config-args> <pkg-config-args>

    # If task manifest refers to any runtime tests, examples, or
    # benchmarks packages:
    #
    {
      # bpkg.test-separate-installed.configure.add (
      #   : bpkg.configure.add)
      #
      bpkg -v add -d <host-conf> <repository-url>

      # bpkg.test-separate-installed.configure.fetch (
      #   : bpkg.configure.fetch)
      #
      bpkg -v fetch -d <host-conf> --trust <repository-fp>
    }

    # If task manifest refers to any build-time tests, examples, or
    # benchmarks packages:
    #
    {
      # bpkg.test-separate-installed.create (
      #   bpkg.test-separate-installed.create_for_host :
      #     bpkg.test-separate-installed.create)
      #
      bpkg -V create -d <target-conf> <env-modules> \
           <env-config-args> <tgt-config-args> <pkg-config-args>

      # [bpkg.test-separate-installed.create]
      #
      b -V create(<module-conf>, cc) \
        config.config.load=~build2-no-warnings

      bpkg -v create --existing --type build2 -d <module-conf>

      # [bpkg.test-separate-installed.link]
```

```
      #
      bpkg -v link -d <target-conf> <host-conf>
      bpkg -v link -d <target-conf> <module-conf>
      bpkg -v link -d <host-conf>   <module-conf>

      # bpkg.test-separate-installed.configure.add (
      #   : bpkg.configure.add)
      #
      bpkg -v add -d <target-conf> <repository-url>

      # bpkg.test-separate-installed.configure.fetch (
      #   : bpkg.configure.fetch)
      #
      bpkg -v fetch -d <target-conf> --trust <repository-fp>
    }

    # bpkg.test-separate-installed.configure.build (
    #   bpkg.global.configure.build,
    #   (bpkg.test-separate-installed.configure.build_for_host :
    #      bpkg.test-separate-installed.configure.build))
    #
    # Note that any of the runtime or build-time tests related parts
    # (but not both) may be omitted.
    #
    bpkg -v build --configure-only \
    <env-config-args> <tgt-config-args> <pkg-config-args> \
    \
    ({ --config-name <host-conf> [<test-config-vars>] }+ \
     <runtime-test-package-name>[ <test-version-constraint>])... \
    \
    ({ --config-name <target-conf> [<test-config-vars>] }+ \
     <buildtime-test-package-name>[ <test-version-constraint>])... \
    \
    ?sys:<package-name>/<package-version> \
    \
    [?sys:<dependency-name>[ <dependency-version-constraint>]...]

    # For each tests, examples, or benchmarks package referred
    # to by the task manifest:
    #
    {
      # bpkg.test-separate-installed.update ( : bpkg.update)
      #
      bpkg -v update <package-name>

      # bpkg.test-separate-installed.test ( : bpkg.test)
      #
      bpkg -v test <package-name>
    }
  }
}

# If the main package is installed from the binary distribution package:
#
{
  # If <distribution> is 'debian':
  #
  {
    # bbot.sys-uninstall.apt-get.remove
    #
    sudo apt-get remove <distribution-package-name>...
  }
```

```
  #
  # Otherwise, if <distribution> is 'fedora':
  #
  {
    # bbot.sys-uninstall.dnf.remove
    #
    sudo dnf remove <distribution-package-name>...
  }
  #
  # Otherwise, if <distribution> is 'archive':
  #
  {
    # Noop.
  }
}

# If the main package is installed from source:
#
{
  # bpkg.uninstall
  #
  bpkg -v uninstall -d <install-conf> <package-name>
}

# If the install operation is supported by the package and
# bbot.bindist.upload step is enabled:
#
{
  # Move the generated binary distribution files to the
  # upload/bindist/<distribution>/ directory.
}

# If bbot.upload step is enabled and upload/ directory is not empty:
#
{
  # bbot.upload.tar.create
  #
  tar -cf upload.tar upload/

  # bbot.upload.tar.list
  #
  tar -tf upload.tar upload/
}

# end
#
# This step id can only be used as a breakpoint.
```

Worker script for `module` packages:

```
# If configuration is self-hosted:
#
{
  # bpkg.create (bpkg.module.create)
  #
  b -V create(<module-conf>, <env-modules>) config.config.load=~build2 \
    <env-config-args> <tgt-config-args> <pkg-config-args>
  bpkg -v create --existing --type build2 -d <module-conf>
}
#
# Otherwise:
```

```
#
{
  # [bpkg.create]
  #
  b -V create(<module-conf>, cc) config.config.load=~build2-no-warnings
  bpkg -v create --existing --type build2 -d <module-conf>
}

# bpkg.configure.add
#
bpkg -v add -d <module-conf> <repository-url>

# bpkg.configure.fetch
#
bpkg -v fetch -d <module-conf> --trust <repository-fp>

# If configuration is self-hosted and config.install.root is specified:
#
{
  # bpkg.create (bpkg.module.create)
  #
  b -V create(<install-conf>, <env-modules>) \
    config.config.load=~build2 \
    <env-config-args> <tgt-config-args> <pkg-config-args>
  bpkg -v create --existing --type build2 -d <install-conf>

  # bpkg.configure.add
  #
  bpkg -v add -d <install-conf> <repository-url>

  # bpkg.configure.fetch
  #
  bpkg -v fetch -d <install-conf> --trust <repository-fp>
}

# If task manifest refers to any (build-time) tests, examples, or
# benchmarks packages:
#
{
  # bpkg.create (bpkg.target.create : b.create, bpkg.create)
  #
  bpkg -V create -d <target-conf> <env-modules> \
       <env-config-args> <tgt-config-args> <pkg-config-args>

  # [bpkg.create]
  #
  b -V create(<host-conf>, cc) config.config.load=~host-no-warnings
  bpkg -v create --existing --type host -d <host-conf>

  # [bpkg.link]
  #
  bpkg -v link -d <target-conf> <host-conf>
  bpkg -v link -d <target-conf> <module-conf>
  bpkg -v link -d <host-conf>   <module-conf>

  # bpkg.configure.add
  #
  bpkg -v add -d <target-conf> <repository-url>

  # bpkg.configure.fetch
  #
  bpkg -v fetch -d <target-conf> --trust <repository-fp>
```

```
}

# bpkg.configure.build (bpkg.global.configure.build)
#
# Notes:
#
# - Some parts may be omitted.
#
# - Parts related to different configurations have different prefix
#   step ids:
#
#   bpkg.module.configure.build for <module-uuid>
#   bpkg.target.configure.build for <install-uuid>
#   bpkg.target.configure.build for <target-uuid>
#
# - All parts have the same fallback step ids: b.configure and
#   bpkg.configure.build.
#
bpkg -v build --configure-only \
<env-config-args> <tgt-config-args> <pkg-config-args> \
[<pkg-config-opts>] \
\
{ --config-uuid <module-uuid> \
  <env-config-args> <tgt-config-args> <pkg-config-args> \
  [<pkg-config-vars>] }+ \
<package-name>/<package-version> \
\
{ --config-uuid <install-uuid> \
  <env-config-args> <tgt-config-args> <pkg-config-args> \
  [<pkg-config-vars>] }+ \
<package-name>/<package-version> \
\
({ --config-uuid <target-uuid> \
   <env-config-args> <tgt-config-args> <pkg-config-args> \
   [<test-config-vars>] }+ \
 <buildtime-test-package-name>[ <test-version-constraint>])... \
\
({ --config-uuid <host-uuid> [--config-uuid <install-uuid>] \
   [<dep-config-vars>] }+ \
 (?|sys:)<dependency-name>[ <dependency-version-constraint>])... \
\
[?sys:<dependency-name>[ <dependency-version-constraint>]...]

# bpkg.update
#
bpkg -v update -d <module-conf> <package-name>

# If the test operation is supported by the package:
#
{
  # bpkg.test
  #
  bpkg -v test -d <module-conf> <package-name>
}

# For each (build-time) tests, examples, or benchmarks package referred
# to by the task manifest:
#
{
  # bpkg.test-separate.update ( : bpkg.update)
  #
  bpkg -v update -d <target-conf> <package-name>
```

```
  # bpkg.test-separate.test ( : bpkg.test)
  #
  bpkg -v test -d <target-conf> <package-name>
}

# If configuration is self-hosted, the install operation is supported
# by the package, config.install.root is specified, and no
# bpkg.bindist.{debian,fedora,archive} step is enabled:
#
{
  # bpkg.install
  #
  bpkg -v install -d <install-conf> <package-name>

  # If bbot.install.ldconfig step is enabled:
  #
  {
    # bbot.install.ldconfig
    #
    sudo ldconfig
  }
}

# If configuration is self-hosted, the install operation is supported
# by the package, and bpkg.bindist.{debian,fedora,archive} step is
# enabled:
#
{
  # bpkg.bindist.{debian,fedora,archive}
  #
  bpkg -v bindist --distribution <distribution> \
       <env-config-args> <tgt-config-args> <pkg-config-args> \
       <package-name>
}

# If the install operation is supported by the package and
# bbot.sys-install step is enabled:
#
{
  # If <distribution> is 'debian':
  #
  {
    # bbot.sys-install.apt-get.update
    #
    sudo apt-get update

    # bbot.sys-install.apt-get.install
    #
    sudo apt-get install <distribution-package-file>...
  }
  #
  # Otherwise, if <distribution> is 'fedora':
  #
  {
    # bbot.sys-install.dnf.install
    #
    sudo dnf install <distribution-package-file>...
  }
  #
  # Otherwise, if <distribution> is 'archive':
  #
```

```
  {
    # For each package file:
    #
    {
      # bbot.sys-install.tar.extract
      #
      [sudo] tar -xf <distribution-package-file> \
                 <env-config-args> <tgt-config-args> <pkg-config-args>
    }

    # If bbot.sys-install.ldconfig step is enabled:
    #
    {
      # bbot.sys-install.ldconfig
      #
      sudo ldconfig
    }
  }
}

# If the main package is installed either from source or from the
# binary distribution package:
#
{
  # If task manifest refers to any (build-time) tests, examples, or
  # benchmarks packages:
  #
  {
    # [bpkg.test-separate-installed.create]
    #
    b -V create(<module-conf>, cc) \
      config.config.load=~build2-no-warnings

    bpkg -v create --existing --type build2 -d <module-conf>

    # bpkg.test-separate-installed.create (
    #   bpkg.test-separate-installed.create_for_module :
    #     bpkg.test-separate-installed.create)
    #
    bpkg -V create -d <target-conf> <env-modules> \
         <env-config-args> <tgt-config-args> <pkg-config-args>

    # bpkg.test-separate-installed.create (
    #   bpkg.test-separate-installed.create_for_module :
    #     bpkg.test-separate-installed.create)
    #
    bpkg -V create --type host -d <host-conf> <env-modules> \
         <env-config-args> <tgt-config-args> <pkg-config-args>

    # [bpkg.test-separate-installed.link]
    #
    bpkg -v link -d <target-conf> <host-conf>
    bpkg -v link -d <target-conf> <module-conf>
    bpkg -v link -d <host-conf>   <module-conf>

    # bpkg.test-separate-installed.configure.add (
    #   : bpkg.configure.add)
    #
    bpkg -v add -d <target-conf> <repository-url>

    # bpkg.test-separate-installed.configure.fetch (
    #   : bpkg.configure.fetch)
```

```
    #
    bpkg -v fetch -d <target-conf> --trust <repository-fp>

    # bpkg.test-separate-installed.configure.build (
    #   bpkg.global.configure.build,
    #   (bpkg.test-separate-installed.configure.build_for_module :
    #      bpkg.test-separate-installed.configure.build))
    #
    bpkg -v build --configure-only \
    <env-config-args> <tgt-config-args> <pkg-config-args> \
    \
    ({ --config-name <target-conf> [<test-config-vars>] }+ \
     <buildtime-test-package-name>[ <test-version-constraint>])... \
    \
    ?sys:<package-name>/<package-version> \
    \
    [?sys:<dependency-name>[ <dependency-version-constraint>]...]

    # For each (build-time) tests, examples, or benchmarks package
    # referred to by the task manifest:
    #
    {
      # bpkg.test-separate-installed.update ( : bpkg.update)
      #
      bpkg -v update -d <target-conf> <package-name>

      # bpkg.test-separate-installed.test ( : bpkg.test)
      #
      bpkg -v test -d <target-conf> <package-name>
    }
  }
}

# If the main package is installed from the binary distribution package:
#
{
  # If <distribution> is 'debian':
  #
  {
    # bbot.sys-uninstall.apt-get.remove
    #
    sudo apt-get remove <distribution-package-name>...
  }
  #
  # Otherwise, if <distribution> is 'fedora':
  #
  {
    # bbot.sys-uninstall.dnf.remove
    #
    sudo dnf remove <distribution-package-name>...
  }
  #
  # Otherwise, if <distribution> is 'archive':
  #
  {
    # Noop.
  }
}

# If the main package is installed from source:
#
{
```

```
  # bpkg.uninstall
  #
  bpkg -v uninstall -d <install-conf> <package-name>
}

# If the install operation is supported by the package and
# bbot.bindist.upload step is enabled:
#
{
  # Move the generated binary distribution files to the
  # upload/bindist/<distribution>/ directory.
}

# If bbot.upload step is enabled and upload/ directory is not empty:
#
{
  # bbot.upload.tar.create
  #
  tar -cf upload.tar upload/

  # bbot.upload.tar.list
  #
  tar -tf upload.tar upload/
}

# end
#
# This step id can only be used as a breakpoint.
```

For details on configuring and testing installation refer to Controller Logic.

If a primary or test package comes from a version control-based repository, then its `dist` meta-operation is also tested as a part of the `bpkg[.*].configure.build` steps by re-distributing the source directory in the load distribution mode after configuration.

If the build is interactive, then the worker pauses its execution at the specified breakpoint and prompts the user whether to continue or abort the execution. If the breakpoint is a step id, then the worker pauses prior to executing every command of the specified step. Otherwise, the breakpoint denotes the result status and the worker pauses if the command results with the specified or more critical status (see Result Manifest).

As an example, the following POSIX shell script can be used to setup the environment for building C and C++ packages with GCC 9 on most Linux distributions.

```
#!/bin/sh

# Environment setup script for C/C++ compilation with GCC 9.
#
# $1  - target
# $2  - bbot executable
# $3+ - bbot options

set -e # Exit on errors.

mode=
case "$1" in
  x86_64-*)
    #mode=-m64
```

```
    ;;
  i?86-*)
    mode=-m32
    ;;
  *)
    echo "unknown target: '$1'" 1>&2
    exit 1
    ;;
esac
shift

exec "$@" cc config.c="gcc-9 $mode" config.cxx="g++-9 $mode"
```

2.9.1 Bindist Result Manifest

At the `bbot.bindist.upload` step the worker also creates the `bindist-result.json` and `bindist-result.manifest` files in the `upload/bindist/<distribution>/` directory, next to the generated binary distribution package files. The `bindist-result.json` file contains the structured JSON output of the **`bpkg-pkg-bindist(1)`** command. The `bindist-result.manifest` file contains the subset of the information from `bindist-result.json`. Specifically, it starts with the binary distribution package header manifest followed by a list of package file manifests. The manifest values are:

```
distribution:
architecture:
os-release-name-id:
os-release-version-id:
package-name:
package-version:
[package-system-version]:

package-file-type:
package-file-path:
[package-file-system-name]:
```

The manifest values derive from the corresponding JSON object values and preserve their semantics. The only differences are that the `os-release-version-id` value may not be absent and the `package-file-path` values are relative to the `upload/bindist/<distribution>/` directory and are in the POSIX representation. See **`bpkg-pkg-bindist(1)`** for the JSON values semantics.

2.10 Controller Logic

A `bbot` controller that issues own build tasks maps available build machines (as reported by agents) to *build target configurations* according to the `buildtab` configuration file. Blank lines and lines that start with # are ignored. All other lines in this file have the following format:

```
<machine-pattern> <target-config> <target>[/<environment>] <classes> [<tgt-config-arg>]* [<warning-regex>]*

<tgt-config-arg> = [[+|-]<prefix>:](<variable>|<option>) | \
                   (+|-)<prefix>:
<prefix> = <tool>[.<cfg-type>][.<phase>][.<operation>[.<command>]]
```

Where <machine-pattern> is filesystem wildcard pattern that is matched against available machine names, <target-config> is the target configuration name, <target> is the build target, optional <environment> is the build environment name, <classes> is a space-separated list of configuration classes that is matched against the package configuration *-builds values, optional <tgt-config-arg> list is additional configuration options and variables, and optional <warning-regex> list is additional regular expressions that should be used to detect warnings in the logs.

The build target configurations can belong to multiple classes with their names reflecting some common configuration aspects, such as the operating system, compiler, build options, etc. Predefined class names are default, all, hidden, none, host, and build2. The default target configurations are built by default. A configuration must also belong to the all, hidden, or some special-purpose configuration class. The latter is intended for testing some optional functionality which packages are not expected to provide normally (for example, relocatable installation). A configuration that is self-hosted must also belong to the host class and, if it is also self-hosted for build system modules, to the build2 class. Valid custom class names must contain only alpha-numeric characters, _, +, -, and ., except as the first character for the last three. Class names that start with _ are reserved for the future hidden/special class functionality.

Regular expressions must start with ~, to be distinguished from target configuration options and variables. Note that the <tgt-config-arg> and <warning-regex> lists have the same quoting semantics as in the target-config and the warning-regex value in the build task manifest. The matched machine name, the target, the environment name, configuration options/variables, and regular expressions are included into the build task manifest.

Values in the <tgt-config-arg> list can be optionally prefixed with the *step id* or a leading portion thereof to restrict it to a specific step, operation, phase, or tool in the *worker script* (see Worker Logic). The prefix can optionally begin with the + or - character (in this case the argument can be omitted) to enable or disable the respective step. The steps which can be enabled or disabled are:

```
bpkg.update
bpkg.test
bpkg.test-separate.update
bpkg.test-separate.test

# Disabled if bpkg.bindist.* is enabled.
#
bpkg.install

# Disabled by default.
#
bbot.install.ldconfig

# Disabled by default.
#
bpkg.bindist.{debian,fedora,archive}

# Disabled if bpkg.bindist.* is disabled.
#
bbot.sys-install
```

```
# Disabled by default.
#
bbot.sys-install.ldconfig

b.test-installed.test
bpkg.test-separate-installed.update
bpkg.test-separate-installed.test

# Disabled by default.
#
bbot.bindist.upload

bbot.upload
```

Note that the `bpkg.bindist.*` steps are mutually exclusive and only the last step status change via the `(+|-)bpkg.bindist.*` prefix is considered.

Unprefixed values only apply to the `*.create[_for_*]` steps. Note that options with values can only be specified using the single argument notation. For example:

```
bpkg:--fetch-timeout=600                \
bpkg.configure.fetch:--fetch-timeout=60 \
+bpkg.bindist.debian:                   \
b:-j1
```

Note that each machine name is matched against every pattern and all the patterns that match produce a target configuration. If a machine does not match any pattern, then it is ignored (meaning that this controller is not interested in testing its packages with this machine). If multiple machines match the same pattern, then only a single target configuration using any of the machines is produced (meaning that this controller considers these machines equivalent).

As an example, let's say we have a machine named `windows_10-vc_14.3`. If we wanted to test both 32 and 64-bit as well as debug and optimized builds, then we could have generated the following target configurations:

```
windows*-msvc_14* windows-msvc_14-Z7 i686-microsoft-win32-msvc14.0 "all default msvc i686 debug" config.cc.coptions=/Z7 config.cc.loptions=/DEBUG ~"warning C4\d{3}: "
windows*-msvc_14* windows-msvc_14-O2 i686-microsoft-win32-msvc14.0 "all default msvc i686 optimized" config.cc.coptions=/O2 ~"warning C4\d{3}: "
windows*-msvc_14* windows-msvc_14-Z7 x86_64-microsoft-win32-msvc14.0 "all default msvc x86_64 debug" config.cc.coptions=/Z7 config.cc.loptions=/DEBUG ~"warning C4\d{3}: "
windows*-msvc_14* windows-msvc_14-O2 x86_64-microsoft-win32-msvc14.0 "all default msvc x86_64 optimized" config.cc.coptions=/O2 ~"warning C4\d{3}: "
```

In the above example we could handle both `i686` and `x86_64` architectures with the same machine but this may not always be possible and we may have to use different machines for different configuration/target combinations. For example:

```
x86_64_linux_debian_11*-gcc_12.2  linux_debian_11-gcc_12.2 i686-linux-gnu   ...

x86_64_linux_debian_11*-gcc_12.2  linux_debian_11-gcc_12.2 x86_64-linux-gnu ...

aarch64_linux_debian_11*-gcc_12.2 linux_debian_11-gcc_12.2 aarch64-linux-gnu ...
```

As another example, let's say we have `linux_fedora_25-gcc_6` and `linux_ubuntu_16.04-gcc_6`. If all we cared about is testing GCC 6 64-bit builds on Linux, then our target configurations could look like this:

```
linux*-gcc_6 linux-gcc_6-g x86_64-linux-gnu "all default gcc debug" config.cc.coptions=-g

linux*-gcc_6 linux-gcc_6-O3 x86_64-linux-gnu "all default gcc optimized" config.cc.coptions=-O3
```

A build target configuration class can derive from another class in which case target configurations that belong to the derived class are treated as also belonging to the base class (or classes, recursively). The derived and base class names are separated with `:` (no leading or trailing spaces allowed) and the base must be present in the first mentioning of the derived class. For example:

```
linux*-gcc_6 linux-gcc_6-g x86_64-linux-gnu "all gcc-6+ debug" config.cc.coptions=-g

linux*-gcc_6 linux-gcc_6-O3 x86_64-linux-gnu "all gcc-6+ optimized" config.cc.coptions=-O3

linux*-gcc_7 linux-gcc_7-g x86_64-linux-gnu "all gcc-7+:gcc-6+ debug" config.cc.coptions=-g

linux*-gcc_7 linux-gcc_7-O3 x86_64-linux-gnu "all gcc-7+ optimized" config.cc.coptions=-O3
```

A machine pattern consisting of a single `-` is a placeholder entry. Everything about a placeholder is ignored except for the class inheritance information. Note, however, that while all other information is ignored, the configuration name and target must be present but can also be `-`. For example:

```
linux*-gcc_6 linux-gcc_6 x86_64-linux-gnu "all gcc-6+       "
-            -           -                "    gcc-7+:gcc-6+"
linux*-gcc_8 linux-gcc_8 x86_64-linux-gnu "all gcc-8+:gcc-7+"
```

If the `<tgt-config-arg>` list contains the `config.install.root` variable that applies to the `bpkg.target.create` or, as a fallback, `b.create` or `bpkg.create` steps, then in addition to building and possibly running tests, the `bbot` worker will also test installing and uninstalling each package (unless replaced with the `bbot.sys-install` step). Furthermore, if the package contains subprojects that support the test operation and/or refers to other packages via the `tests`, `examples`, or `benchmarks` manifest values which are not excluded by the `bbot` controller, then the worker will additionally build such subprojects/packages against the installation (created either from source or from the binary distribution package) and run their tests (test installed and test separate installed phases).

Two types of installations can be tested: *system* and *private*. A system installation uses a well-known location, such as `/usr` or `/usr/local`, that will be searched by the compiler toolchain by default. A private installation uses a private directory, such as `/opt`, that will have to be explicitly mentioned to the compiler. While the system installation is usually preferable, it may not be always usable because of the potential conflicts with the already installed software, for example, by the system package manager.

As an example, the following two target configurations could be used to test system and private installations:

```
linux*-gcc* linux-gcc-sysinstall x86_64-linux-gnu "all default gcc" config.install.root=/usr config.install.sudo=sudo

linux*-gcc* linux-gcc-prvinstall x86_64-linux-gnu "all default gcc" config.install.root=/tmp/install config.cc.poptions=-I/tmp/install/include config.cc.loptions=-L/tmp/install/lib config.bin.rpath=/tmp/install/lib
```

Note also that while building and running tests against the installation created either from source or from the archive distribution package the worker makes the `bin` subdirectory of `config.install.root` the first entry in the `PATH` environment variable, except for build system modules which supposedly don't install any executables. As was mentioned earlier, normally the `config.install.root` variable is expected to be prefixed with the `bpkg.target.create` or, as a fallback, `b.create` or `bpkg.create` step ids. However, for testing of the relocatable installations it can be desirable to extract the archive distribution package content at the `bbot.sys-install.tar.extract` step into a different installation directory. If that's the case, then this directory needs to also be specified as `bbot.sys-install:config.install.root`. If specified, this directory will be preferred as a base for forming the `bin/` directory path.

The `bbot` controller normally issues the build task by picking an unbuilt package configuration and one of the produced (via the machine names match) target configurations, which is not excluded from building due to this package configuration `*-builds`, `*-build-include`, and `*-build-exclude` manifest values.