BitBake User Manual

Copyright © 2004, 2005, 2006, 2011 Chris Larson, Phil Blundell, Richard Purdie

This work is licensed under the Creative Commons Attribution License. To view a copy of this license, visit http://creativecommons.org/-licenses/by/2.5/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Contents

Chapter 1

Introduction

1.1 Overview

BitBake is, at its simplest, a tool for executing tasks and managing metadata. As such, its similarities to GNU make and other build tools are readily apparent. It was inspired by Portage, the package management system used by the Gentoo Linux distribution. BitBake is the basis of the OpenEmbedded project, which is being used to build and maintain a number of embedded Linux distributions/projects such as Angstrom and the Yocto project.

1.2 Background and goals

Prior to BitBake, no other build tool adequately met the needs of an aspiring embedded Linux distribution. All of the buildsystems used by traditional desktop Linux distributions lacked important functionality, and none of the ad-hoc *buildroot* systems, prevalent in the embedded space, were scalable or maintainable.

Some important original goals for BitBake were:

- Handle crosscompilation.
- Handle interpackage dependencies (build time on target architecture, build time on native architecture, and runtime).
- Support running any number of tasks within a given package, including, but not limited to, fetching upstream sources, unpacking them, patching them, configuring them, et cetera.
- Must be Linux distribution agnostic (both build and target).
- Must be architecture agnostic
- Must support multiple build and target operating systems (including Cygwin, the BSDs, etc).
- Must be able to be self contained, rather than tightly integrated into the build machine's root filesystem.
- There must be a way to handle conditional metadata (on target architecture, operating system, distribution, machine).
- It must be easy for the person using the tools to supply their own local metadata and packages to operate against.
- Must make it easy to collaborate between multiple projects using BitBake for their builds.
- Should provide an inheritance mechanism to share common metadata between many packages.

Over time it has become apparent that some further requirements were necessary:

- Handle variants of a base recipe (native, sdk, multilib).

- Able to split metadata into layers and allow layers to override each other.
- Allow representation of a given set of input variables to a task as a checksum.
- based on that checksum, allow acceleration of builds with prebuilt components.

BitBake satisfies all the original requirements and many more with extensions being made to the basic functionality to reflect the additionl requirements. Flexibility and power have always been the priorities. It is highly extensible, supporting embedded Python code and execution of any arbitrary tasks.

Chapter 2

Metadata

2.1 Description

BitBake metadata can be classified into 3 major areas:

- Configuration Files
- .bb Files
- Classes

What follows are a large number of examples of BitBake metadata. Any syntax which isn't supported in any of the aforementioned areas will be documented as such.

2.1.1 Basic variable setting

```
VARIABLE = "value"
```

In this example, `VARIABLE` is `value`.

2.1.2 Variable expansion

BitBake supports variables referencing one another's contents using a syntax which is similar to shell scripting

```
A = "aval"
B = "pre${A}post"
```

This results in `A` containing `aval` and `B` containing `preavalpost`.

2.1.3 Setting a default value (?=)

```
A ?= "aval"
```

If `A` is set before the above is called, it will retain its previous value. If `A` is unset prior to the above call, `A` will be set to `aval`. Note that this assignment is immediate, so if there are multiple ?= assignments to a single variable, the first of those will be used.

2.1.4 Setting a weak default value (??=)

```
A ??= "somevalue"
A ??= "someothervalue"
```

If A is set before the above, it will retain that value. If A is unset prior to the above, A will be set to someothervalue. This is a lazy/weak assignment in that the assignment does not occur until the end of the parsing process, so that the last, rather than the first, ??= assignment to a given variable will be used. Any other setting of A using = or ?= will however override the value set with ??=

2.1.5 Immediate variable expansion (:=)

:= results in a variable's contents being expanded immediately, rather than when the variable is actually used.

```
T = "123"
A := "${B} ${A} test ${T}"
T = "456"
B = "${T} bval"

C = "cval"
C := "${C}append"
```

In that example, A would contain test 123, B would contain 456 bval, and C would be cvalappend.

2.1.6 Appending (+=) and prepending (=+)

```
B = "bval"
B += "additionaldata"
C = "cval"
C =+ "test"
```

In this example, B is now bval additionaldata and C is test cval.

2.1.7 Appending (.=) and prepending (=.) without spaces

```
B = "bval"
B .= "additionaldata"
C = "cval"
C =. "test"
```

In this example, B is now bvaladditionaldata and C is testcval. In contrast to the above appending and prepending operators, no additional space will be introduced.

2.1.8 Conditional metadata set

OVERRIDES is a ":" separated variable containing each item you want to satisfy conditions. So, if you have a variable which is conditional on "arm", and "arm" is in OVERRIDES, then the "arm" specific version of the variable is used rather than the non-conditional version. Example:

```
OVERRIDES = "architecture:os:machine"
TEST = "defaultvalue"
TEST_os = "osspecificvalue"
TEST_condnotinoverrides = "othercondvalue"
```

In this example, TEST would be osspecificvalue, due to the condition "os" being in OVERRIDES.

2.1.9 Conditional appending

BitBake also supports appending and prepending to variables based on whether something is in OVERRIDES. Example:

```
DEPENDS = "glibc ncurses"
OVERRIDES = "machine:local"
DEPENDS_append_machine = " libmad"
```

In this example, `DEPENDS` is set to `glibc ncurses libmad`.

2.1.10 Inclusion

Next, there is the `include` directive, which causes BitBake to parse whatever file you specify, and insert it at that location, which is not unlike **make**. However, if the path specified on the `include` line is a relative path, BitBake will locate the first one it can find within `BBPATH`.

2.1.11 Requiring inclusion

In contrast to the `include` directive, `require` will raise an ParseError if the file to be included cannot be found. Otherwise it will behave just like the `include` directive.

2.1.12 Python variable expansion

```
DATE = "${@time.strftime('%Y%m%d',time.gmtime())}"
```

This would result in the `DATE` variable containing today's date.

2.1.13 Defining executable metadata

NOTE: This is only supported in .bb and .bbclass files.

```
do_mytask () {
    echo "Hello, world!"
}
```

This is essentially identical to setting a variable, except that this variable happens to be executable shell code.

```
python do_printdate () {
    import time
    print time.strftime('%Y%m%d', time.gmtime())
}
```

This is the similar to the previous, but flags it as Python so that BitBake knows it is Python code.

2.1.14 Defining Python functions into the global Python namespace

NOTE: This is only supported in .bb and .bbclass files.

```
def get_depends(bb, d):
    if d.getVar('SOMECONDITION', True):
        return "dependencywithcond"
    else:
        return "dependency"

SOMECONDITION = "1"
DEPENDS = "${@get_depends(bb, d)}"
```

This would result in `DEPENDS` containing `dependencywithcond`.

2.1.15 Variable flags

Variables can have associated flags which provide a way of tagging extra information onto a variable. Several flags are used internally by BitBake but they can be used externally too if needed. The standard operations mentioned above also work on flags.

```
VARIABLE[SOMEFLAG] = "value"
```

In this example, `VARIABLE` has a flag, `SOMEFLAG` which is set to `value`.

2.1.16 Inheritance

NOTE: This is only supported in .bb and .bbclass files.

The `inherit` directive is a means of specifying what classes of functionality your .bb requires. It is a rudimentary form of inheritance. For example, you can easily abstract out the tasks involved in building a package that uses autoconf and automake, and put that into a bbclass for your packages to make use of. A given bbclass is located by searching for classes/filename.bbclass in `BBPATH`, where filename is what you inherited.

2.1.17 Tasks

NOTE: This is only supported in .bb and .bbclass files.

In BitBake, each step that needs to be run for a given .bb is known as a task. There is a command `addtask` to add new tasks (must be a defined Python executable metadata and must start with "do_") and describe intertask dependencies.

```
python do_printdate () {
    import time
    print time.strftime('%Y%m%d', time.gmtime())
}

addtask printdate before do_build
```

This defines the necessary Python function and adds it as a task which is now a dependency of do_build, the default task. If anyone executes the do_build task, that will result in do_printdate being run first.

2.1.18 Task Flags

Tasks support a number of flags which control various functionality of the task. These are as follows:

'dirs' - directories which should be created before the task runs

'cleandirs' - directories which should created before the task runs but should be empty

'noexec' - marks the tasks as being empty and no execution required. These are used as dependency placeholders or used when added tasks need to be subsequently disabled.

'nostamp' - don't generate a stamp file for a task. This means the task is always rexecuted.

'fakeroot' - this task needs to be run in a fakeroot environment, obtained by adding the variables in FAKEROOTENV to the environment.

'umask' - the umask to run the task under.

For the 'deptask', 'rdeptask', 'depends', 'rdepends' and 'recrdeptask' flags please see the dependencies section.

2.1.19 Events

NOTE: This is only supported in .bb and .bbclass files.

BitBake allows installation of event handlers. Events are triggered at certain points during operation, such as the beginning of operation against a given .bb, the start of a given task, task failure, task success, et cetera. The intent is to make it easy to do things like email notification on build failure.

```
addhandler myclass_eventhandler
python myclass_eventhandler() {
    from bb.event import getName
    from bb import data

    print("The name of the Event is %s" % getName(e))
    print("The file we run for is %s" % data.getVar('FILE', e.data, True))
}
```

This event handler gets called every time an event is triggered. A global variable `e` is defined. `e`.data contains an instance of bb.data. With the getName(`e`) method one can get the name of the triggered event.

The above event handler prints the name of the event and the content of the `FILE` variable.

2.1.20 Variants

Two BitBake features exist to facilitate the creation of multiple buildable incarnations from a single recipe file.

The first is `BBCLASSEXTEND`. This variable is a space separated list of classes used to "extend" the recipe for each variant. As an example, setting

```
BBCLASSEXTEND = "native"
```

results in a second incarnation of the current recipe being available. This second incarnation will have the "native" class inherited.

The second feature is `BBVERSIONS`. This variable allows a single recipe to build multiple versions of a project from a single recipe file, and allows you to specify conditional metadata (using the `OVERRIDES` mechanism) for a single version, or an optionally named range of versions:

```
BBVERSIONS = "1.0 2.0 git"
SRC_URI_git = "git://someurl/somepath.git"
```

```
BBVERSIONS = "1.0.[0-6]:1.0.0+ \
              1.0.[7-9]:1.0.7+"
SRC_URI_append_1.0.7+ = "file://some_patch_which_the_new_versions_need.patch;patch=1"
```

Note that the name of the range will default to the original version of the recipe, so given OE, a recipe file of foo_1.0.0+.bb will default the name of its versions to 1.0.0+. This is useful, as the range name is not only placed into overrides; it's also made available for the metadata to use in the form of the `BPV` variable, for use in file:// search paths (`FILESPATH`).

2.2 Variable interaction: Worked Examples

Despite the documentation of the different forms of variable definition above, it can be hard to work out what happens when variable operators are combined. This section documents some common questions people have regarding the way variables interact.

2.2.1 Override and append ordering

There is often confusion about which order overrides and the various append operators take effect.

```
OVERRIDES = "foo"
A_foo_append = "X"
```

In this case, X is unconditionally appended to the variable `A_foo`. Since foo is an override, A_foo would then replace `A`.

```
OVERRIDES = "foo"
A = "X"
A_append_foo = "Y"
```

In this case, only when foo is in OVERRIDES, Y is appended to the variable `A` so the value of `A` would become XY (NB: no spaces are appended).

```
OVERRIDES = "foo"
A_foo_append = "X"
A_foo_append += "Y"
```

This behaves as per the first case above, but the value of `A` would be "X Y" instead of just "X".

```
A = "1"
A_append = "2"
A_append = "3"
A += "4"
A .= "5"
```

Would ultimately result in `A` taking the value "1 4523" since the _append operator executes at the same time as the expansion of other overrides.

2.2.2 Key Expansion

Key expansion happens at the data store finalisation time just before overrides are expanded.

```
A${B} = "X"
B = "2"
A2 = "Y"
```

So in this case `A2` would take the value of "X".

2.3 Dependency handling

BitBake handles dependencies at the task level since to allow for efficient operation with multiple processed executing in parallel. A robust method of specifying task dependencies is therefore needed.

2.3.1 Dependencies internal to the .bb file

Where the dependencies are internal to a given .bb file, the dependencies are handled by the previously detailed addtask directive.

2.3.2 Build Dependencies

DEPENDS lists build time dependencies. The 'deptask' flag for tasks is used to signify the task of each item listed in DEPENDS which must have completed before that task can be executed.

```
do_configure[deptask] = "do_populate_staging"
```

means the do_populate_staging task of each item in DEPENDS must have completed before do_configure can execute.

2.3.3 Runtime Dependencies

The PACKAGES variable lists runtime packages and each of these can have RDEPENDS and RRECOMMENDS runtime dependencies. The 'rdeptask' flag for tasks is used to signify the task of each item runtime dependency which must have completed before that task can be executed.

```
do_package_write[rdeptask] = "do_package"
```

means the do_package task of each item in RDEPENDS must have completed before do_package_write can execute.

2.3.4 Recursive Dependencies

These are specified with the 'recrdeptask' flag which is used signify the task(s) of dependencies which must have completed before that task can be executed. It works by looking though the build and runtime dependencies of the current recipe as well as any inter-task dependencies the task has, then adding a dependency on the listed task. It will then recurse through the dependencies of those tasks and so on.

It may be desireable to recurse not just through the dependencies of those tasks but through the build and runtime dependencies of dependent tasks too. If that is the case, the taskname itself should be referenced in the task list, e.g. do_a[recrdeptask] = "do_a do_b".

2.3.5 Inter task

The 'depends' flag for tasks is a more generic form of which allows an interdependency on specific tasks rather than specifying the data in DEPENDS.

```
do_patch[depends] = "quilt-native:do_populate_staging"
```

means the do_populate_staging task of the target quilt-native must have completed before the do_patch can execute.

The 'rdepends' flag works in a similar way but takes targets in the runtime namespace instead of the build time dependency namespace.

2.4 Parsing

2.4.1 Configuration files

The first kind of metadata in BitBake is configuration metadata. This metadata is global, and therefore affects *all* packages and tasks which are executed.

BitBake will first search the current working directory for an optional "conf/bblayers.conf" configuration file. This file is expected to contain a BBLAYERS variable which is a space delimited list of 'layer' directories. For each directory in this list, a "conf/layer.conf" file will be searched for and parsed with the LAYERDIR variable being set to the directory where the layer was found. The idea is these files will setup BBPATH and other variables correctly for a given build directory automatically for the user.

BitBake will then expect to find 'conf/bitbake.conf' somewhere in the user specified `BBPATH`. That configuration file generally has include directives to pull in any other metadata (generally files specific to architecture, machine, *local* and so on).

Only variable definitions and include directives are allowed in .conf files.

2.4.2 Classes

BitBake classes are our rudimentary inheritance mechanism. As briefly mentioned in the metadata introduction, they're parsed when an `inherit` directive is encountered, and they are located in classes/ relative to the directories in `BBPATH`.

2.4.3 .bb files

A BitBake (.bb) file is a logical unit of tasks to be executed. Normally this is a package to be built. Inter-.bb dependencies are obeyed. The files themselves are located via the `BBFILES` variable, which is set to a space separated list of .bb files, and does handle wildcards.

Chapter 3

File download support

3.1 Overview

BitBake provides support to download files this procedure is called fetching and it handled by the fetch and fetch2 modules. At this point the original fetch code is considered to be replaced by fetch2 and this manual only related to the fetch2 codebase.

The SRC_URI is normally used to tell BitBake which files to fetch. The next sections will describe the available fetchers and their options. Each fetcher honors a set of variables and per URI parameters separated by a ";" consisting of a key and a value. The semantics of the variables and parameters are defined by the fetcher. BitBake tries to have consistent semantics between the different fetchers.

The overall fetch process is that first, fetches are attempted from PREMIRRORS. If those don't work, the original SRC_URI is attempted and if that fails, BitBake will fall back to MIRRORS. Cross urls are supported, so its possible to mirror a git repository on an http server as a tarball for example. Some example commonly used mirror definitions are:

```
PREMIRRORS ?= "\
bzr://.*/.*   http://somemirror.org/sources/ \n \
cvs://.*/.*   http://somemirror.org/sources/ \n \
git://.*/.*   http://somemirror.org/sources/ \n \
hg://.*/.*    http://somemirror.org/sources/ \n \
osc://.*/.*   http://somemirror.org/sources/ \n \
p4://.*/.*    http://somemirror.org/sources/ \n \
svk://.*/.*   http://somemirror.org/sources/ \n \
svn://.*/.*   http://somemirror.org/sources/ \n"

MIRRORS =+ "\
ftp://.*/.*      http://somemirror.org/sources/ \n \
http://.*/.*     http://somemirror.org/sources/ \n \
https://.*/.*    http://somemirror.org/sources/ \n"
```

Non-local downloaded output is placed into the directory specified by the `DL_DIR`. For non local archive downloads the code can verify sha256 and md5 checksums for the download to ensure the file has been downloaded correctly. These may be specified either in the form `SRC_URI[md5sum]` for the md5 checksum and `SRC_URI[sha256sum]` for the sha256 checksum or as parameters on the SRC_URI such as SRC_URI="http://example.com/foobar.tar.bz2;md5sum=4a8e0f237e961fd7785d19d07fdb994d". If `BB_STRICT_CHECKSUM` is set, any download without a checksum will trigger an error message. In cases where multiple files are listed in SRC_URI, the name parameter is used assign names to the urls and these are then specified in the checksums in the form SRC_URI[name.sha256sum].

3.2 Local file fetcher

The URN for the local file fetcher is *file*. The filename can be either absolute or relative. If the filename is relative, `FILESPATH` and failing that `FILESDIR` will be used to find the appropriate relative file. The metadata usually extend these variables to include variations of the values in `OVERRIDES`. Single files and complete directories can be specified.

```
SRC_URI= "file://relativefile.patch"
SRC_URI= "file://relativefile.patch;this=ignored"
SRC_URI= "file:///Users/ich/very_important_software"
```

3.3 CVS fetcher

The URN for the CVS fetcher is *cvs*. This fetcher honors the variables CVSDIR, SRCDATE, FETCHCOMMAND_cvs, UPDATECOMMAND_cvs. DL_DIR specifies where a temporary checkout is saved. SRCDATE specifies which date to use when doing the fetching (the special value of "now" will cause the checkout to be updated on every build). FETCHCOMMAND and UPDATECOMMAND specify which executables to use for the CVS checkout or update.

The supported parameters are module, tag, date, method, localdir, rsh and scmdata. The module specifies which module to check out, the tag describes which CVS TAG should be used for the checkout. By default the TAG is empty. A date can be specified to override the SRCDATE of the configuration to checkout a specific date. The special value of "now" will cause the checkout to be updated on every build.method is by default *pserver*. If *ext* is used the rsh parameter will be evaluated and CVS_RSH will be set. Finally, localdir is used to checkout into a special directory relative to CVSDIR.

```
SRC_URI = "cvs://CVSROOT;module=mymodule;tag=some-version;method=ext"
SRC_URI = "cvs://CVSROOT;module=mymodule;date=20060126;localdir=usethat"
```

3.4 HTTP/FTP fetcher

The URNs for the HTTP/FTP fetcher are *http*, *https* and *ftp*. This fetcher honors the variables FETCHCOMMAND_wget. FETCHCOMMAND contains the command used for fetching. “${URI}” and “${FILES}” will be replaced by the URI and basename of the file to be fetched.

```
SRC_URI = "http://oe.handhelds.org/not_there.aac"
SRC_URI = "ftp://oe.handhelds.org/not_there_as_well.aac"
SRC_URI = "ftp://you@oe.handheld.sorg/home/you/secret.plan"
```

3.5 SVN fetcher

The URN for the SVN fetcher is *svn*.

This fetcher honors the variables FETCHCOMMAND_svn, SVNDIR, SRCREV. FETCHCOMMAND contains the subversion command. SRCREV specifies which revision to use when doing the fetching.

The supported parameters are proto, rev and scmdata. proto is the Subversion protocol, rev is the Subversion revision. If scmdata is set to “keep”, the “.svn” directories will be available during compile-time.

```
SRC_URI = "svn://svn.oe.handhelds.org/svn;module=vip;proto=http;rev=667"
SRC_URI = "svn://svn.oe.handhelds.org/svn/;module=opie;proto=svn+ssh;date=20060126"
```

3.6 GIT fetcher

The URN for the GIT Fetcher is *git*.

The variable GITDIR will be used as the base directory where the git tree is cloned to.

The parameters are *tag*, *protocol* and *scmdata*. *tag* is a Git tag, the default is “master”. *protocol* is the Git protocol to use and defaults to “git” if a hostname is set, otherwise its “file”. If *scmdata* is set to “keep”, the “.git” directory will be available during compile-time.

```
SRC_URI = "git://git.oe.handhelds.org/git/vip.git;tag=version-1"
SRC_URI = "git://git.oe.handhelds.org/git/vip.git;protocol=http"
```

Chapter 4

The BitBake command

4.1 Introduction

bitbake is the primary command in the system. It facilitates executing tasks in a single .bb file, or executing a given task on a set of multiple .bb files, accounting for interdependencies amongst them.

4.2 Usage and syntax

```
$ bitbake --help
usage: bitbake [options] [package ...]

Executes the specified task (default is 'build') for a given set of BitBake files.
It expects that BBFILES is defined, which is a space separated list of files to
be executed.  BBFILES does support wildcards.
Default BBFILES are the .bb files in the current directory.

options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -b BUILDFILE, --buildfile=BUILDFILE
                        execute the task against this .bb file, rather than a
                        package from BBFILES.
  -k, --continue        continue as much as possible after an error. While the
                        target that failed, and those that depend on it,
                        cannot be remade, the other dependencies of these
                        targets can be processed all the same.
  -f, --force           force run of specified cmd, regardless of stamp status
  -i, --interactive     drop into the interactive mode also called the BitBake
                        shell.
  -c CMD, --cmd=CMD     Specify task to execute. Note that this only executes
                        the specified task for the providee and the packages
                        it depends on, i.e. 'compile' does not implicitly call
                        stage for the dependencies (IOW: use only if you know
                        what you are doing). Depending on the base.bbclass a
                        listtasks task is defined and will show available
                        tasks
  -r FILE, --read=FILE  read the specified file before bitbake.conf
  -v, --verbose         output more chit-chat to the terminal
  -D, --debug           Increase the debug level. You can specify this more
                        than once.
  -n, --dry-run         don't execute, just go through the motions
  -p, --parse-only      quit after parsing the BB files (developers only)
```

```
  -s, --show-versions   show current and preferred versions of all packages
  -e, --environment     show the global or per-package environment (this is
                        what used to be bbread)
  -g, --graphviz        emit the dependency trees of the specified packages in
                        the dot syntax
  -I IGNORED_DOT_DEPS, --ignore-deps=IGNORED_DOT_DEPS
                        Stop processing at the given list of dependencies when
                        generating dependency graphs. This can help to make
                        the graph more appealing
  -l DEBUG_DOMAINS, --log-domains=DEBUG_DOMAINS
                        Show debug logging for the specified logging domains
  -P, --profile         profile the command and print a report
```

Example 4.1 Executing a task against a single .bb

Executing tasks for a single file is relatively simple. You specify the file in question, and BitBake parses it and executes the specified task (or "build" by default). It obeys intertask dependencies when doing so.
"clean" task:

```
$ bitbake -b blah_1.0.bb -c clean
```

"build" task:

```
$ bitbake -b blah_1.0.bb
```

Example 4.2 Executing tasks against a set of .bb files

There are a number of additional complexities introduced when one wants to manage multiple .bb files. Clearly there needs to be a way to tell BitBake what files are available, and of those, which we want to execute at this time. There also needs to be a way for each .bb to express its dependencies, both for build time and runtime. There must be a way for the user to express their preferences when multiple .bb's provide the same functionality, or when there are multiple versions of a .bb.
The next section, Metadata, outlines how to specify such things.
Note that the bitbake command, when not using --buildfile, accepts a `PROVIDER`, not a filename or anything else. By default, a .bb generally PROVIDES its packagename, packagename-version, and packagename-version-revision.

```
$ bitbake blah
```

```
$ bitbake blah-1.0
```

```
$ bitbake blah-1.0-r0
```

```
$ bitbake -c clean blah
```

```
$ bitbake virtual/whatever
```

```
$ bitbake -c clean virtual/whatever
```

Example 4.3 Generating dependency graphs

BitBake is able to generate dependency graphs using the dot syntax. These graphs can be converted to images using the dot application from Graphviz. Two files will be written into the current working directory, *depends.dot* containing dependency information at the package level and *task-depends.dot* containing a breakdown of the dependencies at the task level. To stop depending on common depends, one can use the `-I depend` to omit these from the graph. This can lead to more readable graphs. This way, `DEPENDS` from inherited classes such as base.bbclass can be removed from the graph.

```
$ bitbake -g blah
```

```
$ bitbake -g -I virtual/whatever -I bloom blah
```

4.3 Special variables

Certain variables affect BitBake operation:

4.3.1 BB_NUMBER_THREADS

The number of threads BitBake should run at once (default: 1).

4.4 Metadata

As you may have seen in the usage information, or in the information about .bb files, the BBFILES variable is how the BitBake tool locates its files. This variable is a space separated list of files that are available, and supports wildcards.

Example 4.4 Setting BBFILES

```
BBFILES = "/path/to/bbfiles/*.bb"
```

With regard to dependencies, it expects the .bb to define a DEPENDS variable, which contains a space separated list of "package names", which themselves are the PN variable. The PN variable is, in general, set to a component of the .bb filename by default.

Example 4.5 Depending on another .bb

a.bb:

```
PN = "package-a"
DEPENDS += "package-b"
```

b.bb:

```
PN = "package-b"
```

Example 4.6 Using PROVIDES

This example shows the usage of the PROVIDES variable, which allows a given .bb to specify what functionality it provides.
package1.bb:

```
PROVIDES += "virtual/package"
```

package2.bb:

```
DEPENDS += "virtual/package"
```

package3.bb:

```
PROVIDES += "virtual/package"
```

As you can see, we have two different .bb's that provide the same functionality (virtual/package). Clearly, there needs to be a way for the person running BitBake to control which of those providers gets used. There is, indeed, such a way.
The following would go into a .conf file, to select package1:

```
PREFERRED_PROVIDER_virtual/package = "package1"
```

Example 4.7 Specifying version preference

When there are multiple "versions" of a given package, BitBake defaults to selecting the most recent version, unless otherwise specified. If the .bb in question has a `DEFAULT_PREFERENCE` set lower than the other .bb's (default is 0), then it will not be selected. This allows the person or persons maintaining the repository of .bb files to specify their preference for the default selected version. In addition, the user can specify their preferred version.
If the first .bb is named `a_1.1.bb`, then the `PN` variable will be set to "a", and the `PV` variable will be set to 1.1.
If we then have an `a_1.2.bb`, BitBake will choose 1.2 by default. However, if we define the following variable in a .conf that BitBake parses, we can change that.

```
PREFERRED_VERSION_a = "1.1"
```

Example 4.8 Using "bbfile collections"

bbfile collections exist to allow the user to have multiple repositories of bbfiles that contain the same exact package. For example, one could easily use them to make one's own local copy of an upstream repository, but with custom modifications that one does not want upstream. Usage:

```
BBFILES = "/stuff/openembedded/*/*.bb /stuff/openembedded.modified/*/*.bb"
BBFILE_COLLECTIONS = "upstream local"
BBFILE_PATTERN_upstream = "^/stuff/openembedded/"
BBFILE_PATTERN_local = "^/stuff/openembedded.modified/"
BBFILE_PRIORITY_upstream = "5"
BBFILE_PRIORITY_local = "10"
```