BitBake User Manual

BitBake Team

BitBake User Manual

by BitBake Team

Table of Contents

List of Examples

Chapter 1. Introduction

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 [http://www.openembedded.org/] project, which is being used to build and maintain a number of embedded Linux distributions, including OpenZaurus and Familiar.

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 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.
- Et cetera...

BitBake satisfies all these and many more. 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

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.

Basic variable setting

```
VARIABLE = "value"
```

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

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`.

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`.

Appending (+=) and prepending (=+)

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

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

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.

Conditional metadata set

OVERRIDES is a ":" seperated 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.

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.

Inclusion

Next, there is the include directive, which causes BitBake to parse in 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.

Requiring Inclusion

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

Python variable expansion

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

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

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.

Defining python functions into the global python namespace

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

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

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

This would result in DEPENDS containing dependencywithcond.

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 rudamentary form of inheritence. 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.oeclass in BBPATH, where filename is what you inherited.

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.

Events

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

BitBake allows to install 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 was to make it easy to do things like email notifications on build failure.

```
addhandler myclass_eventhandler
python myclass_eventhandler() {
   from bb.event import NotHandled, 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)

   return NotHandled
}
```

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.

Parsing

Configuration Files

The first of the classifications of metadata in BitBake is configuration metadata. This metadata is global, and therefore affects *all* packages and tasks which are executed. Currently, BitBake has hardcoded knowledge of a single configuration file. It expects 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.

Classes

BitBake classes are our rudamentary inheritence 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 dirs in `BBPATH`.

.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 seperated list of .bb files, and does handle wildcards.

Chapter 3. File Download support

Overview

BitBake provides support to download files this procedure is called fetching. The SRC_URI is normally used to indicate BitBake which files to fetch. The next sections will describe th available fetchers and the options they have. Each Fetcher honors a set of Variables and a per URI parameters separated by a ";" consisting of a key and a value. The semantic of the Variables and Parameters are defined by the Fetcher. BitBakes tries to have a consistent semantic between the different Fetchers.

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 `FILESDIR` will be used to find the appropriate relative file depending on the `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"
```

CVS File Fetcher

The URN for the CVS Fetcher is *cvs*. This Fetcher honors the variables `DL_DIR`, `SRCDATE`, `FETCHCOMMAND_cvs`, `UPDATECOMMAND_cvs`. `DL_DIRS` 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 should be used when doing the CVS checkout or update.

The supported Parameters are `module`, `tag`, `date`, `method`, `localdir`, `rsh`. 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"
```

HTTP/FTP Fetcher

The URNs for the HTTP/FTP are *http*, *https* and *ftp*. This Fetcher honors the variables `DL_DIR`, `FETCHCOMMAND_wget`, `PREMIRRORS`, `MIRRORS`. The `DL_DIR` defines where to store the fetched file, `FETCHCOMMAND` contains the command used for fetching. "${URI}" and "${FILES}" will be replaced by the uri and basename of the to be fetched file. `PREMIRRORS` will be tried first when fetching a file if that fails the actual file will be tried and finally all `MIRRORS` will be tried.

The only supported Parameter is `md5sum`. After a fetch the `md5sum` of the file will be calculated and the two sums will be compared.

```
SRC_URI = "http://oe.handhelds.org/not_there.aac;md5sum=12343"
SRC_URI = "ftp://oe.handhelds.org/not_there_as_well.aac;md5sum=1234"
```

```
SRC_URI = "ftp://you@oe.handheld.sorg/home/you/secret.plan;md5sum=1234"
```

SVK Fetcher

Currently NOT supported

SVN Fetcher

The URN for the SVN Fetcher is *svn*.

This Fetcher honors the variables `FETCHCOMMAND_svn`, `DL_DIR`, `SRCDATE`. `FETCHCOMMAND` contains the subversion command, `DL_DIR` is the directory where tarballs will be 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).

The supported Parameters are `proto`, `rev`. `proto` is the subversion prototype, `rev` is the subversions revision.

```
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"
```

GIT Fetcher

The URN for the GIT Fetcher is *git*.

The Variables `DL_DIR`, `GITDIR` are used. `DL_DIR` will be used to store the checkedout version. `GITDIR` will be used as the base directory where the git tree is cloned to.

The Parameters are *tag*, *protocol*. *tag* is a git tag, the default is “master”. *protocol* is the git protocol to use and defaults to “rsync”.

```
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. Commands

bbread

bbread is a command for displaying BitBake metadata. When run with no arguments, it has the core parse 'conf/bitbake.conf', as located in BBPATH, and displays that. If you supply a file on the commandline, such as a .bb, then it parses that afterwards, using the aforementioned configuration metadata.

NOTE: the stand a lone bbread command was removed. Instead of bbread use bitbake -e.

bitbake

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.

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 seperated 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 tasks 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)
  -d, --disable-psyco  disable using the psyco just-in-time compiler (not
```

```
                recommended)
 -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
```

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 one goes about specifying 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 [http://www.graphviz.org]. Three files will be written into the current working directory, *depends.dot* containing `DEPENDS` variables, *rdepends.dot* and *alldepends.dot* containing both `DEPENDS` and `RDEPENDS`. 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. E.g. this way `DEPENDS` from inherited classes, e.g. base.bbclass, can be removed from the graph.

```
$ bitbake -g blah
```

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

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 seperated 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 seperated list of "package names", which themselves are the `PN` variable. The `PN` variable is, in general, by default, set to a component of the .bb filename.

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, here there are 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 preferences for the default selected version. In addition, the user can specify their preferences with regard to 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"
```