BitBake User Manual

Richard Purdie, Chris Larson, and Phil Blundell

BitBake Community <bitbake-devel@lists.openembedded.org>

Оглавление

Глава 1. Обзор

В этом документе описана программа BitBake с попыткой максимально абстрагироваться от систем, использующих BitBake, таких как OpenEmbedded (OE) и Yocto Project (YP). В некоторых случаях описание включает варианты и примеры использования с чётким указанием контекста.

1.1. Введение

BitBake представляет собой машину выполнения задач, которая позволяет запускать задачи Python для эффективного выполнения в параллель при работе в контексте ограничений, связанных с зависимостями между задачами. Одно из основных приложений BitBake - система OE создает на этой основе сборки программных стеков Linux для встраиваемых систем с использованием ориентированного на задачи подхода.

В некоторых отношениях BitBake напоминает GNU Make, но имеет ряд существенных отличий.

- BitBake выполняет задачи в соответствии с предоставленными метаданными, которые создают задачу. Метаданные хранятся в заданиях (.bb) и связанных с ними файлах дополнения (.bbappend), конфигурационных (.conf) и включаемых (.inc) файлах, а также в классах (.bbclass), и дают BitBake инструкции о том, какие задачи запускать и как задачи зависят одна от другой.

- BitBake включает библиотеку сборщика для извлечения исходного кода из локальных хранилищ, систем управления SCS[1] или web-сайтов.
- Инструкции для каждой единицы сборки (например, программы) называют заданиями (recipe) и они содержат всю информацию о сборке (зависимости, размещение источников,контрольные суммы, описание и т. п.).
- BitBake поддерживает модель «клиент-сервер» и может использоваться из командной строки или как служба XML-RPC, предоставляя несколько разных пользовательских интерфейсов.

1.2. История и назначение

BitBake исходно был частью проекта OE и его прообразом послужила система управления пакетами Portage из Gentoo Linux. 7 декабря 2004 г. член команды OE Chris Larson разделил проект на две части:

- BitBake в качестве машины исполнения задач;
- OE в качестве набора метаданных, используемых BitBake

Сейчас BitBake служит основой проекта OE, используемого для сборки и поддержки таких дистрибутивов Linux как Angstrom Distribution, а также служащего инструментом сборки проектов Linux, таких как Yocto Project.

До появления BitBake ни один инструмент не соответствовал всем задачам создания дистрибутивов Linux для встраиваемых систем. В стандартных дистрибутивах Linux не хватало функциональности и ни одна из систем на основе Buildroot для встраиваемых платформ не обеспечивала требуемой расширяемости и поддержки.

Ниже перечислены некоторые важные исходные цели BitBake.

- Поддержка кросс-компиляции.
- Обработка зависимостей между пакетами (при сборке на целевой и сборочной системе, а также при работе).
- Поддержка любого числа задач внутри пакета, включая выборку исходных кодов, их распаковку, применение правок (patch), настройка и т. п.
- Независимость от дистрибутива Linux на системе сборки и целевой системе.
- Независимость от архитектуры.
- Поддержка разных сборочных и целевых систем (Cygwin, BSD и т. п.).
- Самодостаточность без тесной интеграции в корневую файловую системы машины сборки.
- Обработка условных метаданных для целевой архитектуры, операционной системы, дистрибутива и машины.
- Простота инструментов для поддержки локальных метаданных и пакетов, с которыми можно работать.
- Простота использования BitBake в совместной работе с разными проектами для их сборки.
- Предоставление механизма наследования для использования пакетами общего набора метаданных.

С течением времени возникли дополнительные требования.

- Обработка вариантов базового задания (например, native, sdk, multilib).
- Разделение метаданных по уровням для расширения и возможности переопределения.
- Возможность представления заданного набора входных переменных задачи контрольной суммой для ускорения сборки при наличии уже собранных компонент.

BitBake удовлетворяет всем исходным требованиям и многим расширениям. Гибкость и возможности всегда были приоритетом. BitBake поддерживает возможности расширения, встроенный код Python и выполнение любых задач.

1.3. Концепции

Программа BitBake написана на языке Python. На верхнем уровне BitBake интерпретирует метаданные, определяет задачи, которые нужно выполнить, и запускает их. Подобно GNU Make, BitBake контролирует сборку программ через файлы, которые называются заданиями (recipe).

BitBake расширяет возможности простых инструментов, таких как GNU Make, позволяя определять более сложные задачи, например, сборку полного дистрибутива Linux для встраиваемой системы.

1.3.1. Задания

Задания BitBake, хранящиеся в файлах .bb, служат базовыми метаданными, обеспечивая BitBake:

- описания пакетов (автор, домашняя страница, лицензия и т. п.);
- версии заданий;
- имеющиеся зависимости (при сборке и работе);
- местоположение исходного кода и способ его извлечения;
- потребность в применении правок, их местоположение и способ применения;
- детали настройки и компиляции исходного кода;
- место установки программ на целевой машине.

[1]Source control system - система управления исходными файлами.

В контексте BitBake или использующей программу системы сборки файлы .bb считаются заданиями. Иногда задания называют пакетами, однако это не совсем корректно, поскольку одно задание может включать множество пакетов.

1.3.2. Файлы конфигурации

Конфигурационные файлы .conf задают переменные, управляющие процессом сборки. Эти файлы делятся на несколько категорий, задающих конфигурацию машины и дистрибутива, опции компиляции, базовые и пользовательские параметры. Основным файлом конфигурации служит bitbake.conf в каталоге conf дерева источников.

1.3.3. Классы

Файлы классов .bbclass содержат информацию, которая может совместно использоваться множеством файлов метаданных. Дерево источников BitBake в настоящее время содержит один файл метаданных классов base.bbclass в каталоге classes. Классы из base.bbclass включаются автоматически для всех заданий и классов и содержат определения стандартных базовых задач, таких как выборка, распаковка, настройка (по умолчанию пуст), компиляция (запускается при наличии Makefile), инсталляция (по умолчанию пуст) и подготовка пакетов (по умолчанию пуст). Эти задачи часто переопределяются или расширяются другими классами, созданными при разработке проектов.

1.3.4. Уровни

Уровни позволяют разделить разные типы настроек. Может показаться заманчивым держать все в одном месте при работе над проектом, однако модульная организация существенно упростит внесение изменений в проект.

Для иллюстрации применения уровней рассмотрим конфигурацию для конкретной целевой машины. Этот тип настройки обычно выделяют в специальный уровень, называемый уровнем BSP[1]. Настройки для машины следует изолировать от заданий и метаданных, поддерживающих, например, новую среду GUI. Однако важно понимать, что уровень BSP может включать машинозависимые дополнения для заданий GUI без вмешательства в этот уровень. Это делается с помощью файлов дополнения BitBake append (.bbappend).

1.3.5. Файлы дополнения

Файлы дополнения .bbappend расширяют и переопределяют информацию имеющегося файла задания. BitBake считает, что каждому файлу дополнения соответствует своё задание, имена дополнения и задания должны иметь общий корень и могут различаться лишь суффиксом (например, formfactor_0.0.bb и formfactor_0.0.bbappend).

Информация в файле дополнения расширяет или переопределяет содержимое соответствующего файла задания. В именах файлов дополнения можно применять шаблон %, позволяющий дополнять множество заданий сразу. Например, файл busybox_1.21.%.bbappend будет соответствовать заданиям busybox_1.21.*x*.bb разных версий.

```
busybox_1.21.1.bb
busybox_1.21.2.bb
busybox_1.21.3.bb
```

Использование символа % ограничено размещением непосредственно перед суффиксом .bbappend. Если задание busybox обновить для busybox_1.3.0.bb, файл дополнения не будет ему соответствовать. Однако можно назвать файл busybox_1.%.bbappend и соответствие будет восстановлено. В более общем случае можно назвать файл busybox_%.bbappend для соответствия любым версиям задания.

1.4. Получение BitBake

1. *Клонирование BitBake* из репозитория Git является рекомендуемым методом установки BitBake. Это позволяет получить файлы исправлений, а также доступ к стабильным ветвям и ветви master. После клонирования BitBake следует использовать последнюю стабильную ветвь, поскольку ветвь master может включать не совсем стабильные изменения. Обычно следует выбирать версию BitBake, соответствующую применяемым метаданным, которые обычно совместимы с прошлыми версиями, но могут не соответствовать новым.

   Для клонирования BitBake служит команда git clone git://git.openembedded.org/bitbake, которая создаст копию репозитория в локальном каталоге bitbake. Если нужно использовать иное имя каталога, его указывают в команде. Например, для размещения локального репозитория в каталоге bbdev следует использовать команду

   ```
   $ git clone git://git.openembedded.org/bitbake bbdev
   ```

2. *Установка из дистрибутива* не рекомендуется, поскольку в большинстве случаев она отстаёт по версии.

3. *Конкретная версия BitBake*, которую можно загрузить из репозитория, указав известную ветвь или выпуск BitBake. Ниже приведен пример загрузки из репозитория BitBake версии 1.17.0

   ```
   $ wget http://git.openembedded.org/bitbake/snapshot/bitbake-1.17.0.tar.gz
   $ tar zxpvf bitbake-1.17.0.tar.gz
   ```

   После распаковки архива файлы будут находиться в каталоге bitbake-1.17.0.

4. *Использование BitBake из выбранной системы сборки.* Вместо выбора и установки отдельных уровней можно установить целиком систему сборки, которая будет включать версию BitBake, протестированную в части совместимости с остальными компонентами.

1.5. Команда bitbake

Команда bitbake является основным интерфейсом BitBake. Здесь описан синтаксис команды и даны примеры.

1.5.1. Использование и синтаксис

Описание синтаксиса можно получить по команде

```
$ bitbake -h
Usage: bitbake [options] [recipename/target recipe:do_task ...]
```

[1]Board Support Package - пакет поддержки плат.

```
    Executes the specified task (default is 'build') for a given set of target recipes (.bb files).
    It is assumed there is a conf/bblayers.conf available in cwd or in BBPATH which
    will provide the layer, BBFILES and other configuration information.

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -b BUILDFILE, --buildfile=BUILDFILE
                        Execute tasks from a specific .bb recipe directly.
                        WARNING: Does not handle any dependencies from other
                        recipes.
  -k, --continue        Continue as much as possible after an error. While the
                        target that failed and anything depending on it cannot
                        be built, as much as possible will be built before
                        stopping.
  -f, --force           Force the specified targets/task to run (invalidating
                        any existing stamp file).
  -c CMD, --cmd=CMD     Specify the task to execute. The exact options
                        available depend on the metadata. Some examples might
                        be 'compile' or 'populate_sysroot' or 'listtasks' may
                        give a list of the tasks available.
  -C INVALIDATE_STAMP, --clear-stamp=INVALIDATE_STAMP
                        Invalidate the stamp for the specified task such as
                        'compile' and then run the default task for the
                        specified target(s).
  -r PREFILE, --read=PREFILE
                        Read the specified file before bitbake.conf.
  -R POSTFILE, --postread=POSTFILE
                        Read the specified file after bitbake.conf.
  -v, --verbose         Enable tracing of shell tasks (with 'set -x'). Also
                        print bb.note(...) messages to stdout (in addition to
                        writing them to ${T}/log.do_<task>).
  -D, --debug           Increase the debug level. You can specify this more
                        than once. -D sets the debug level to 1, where only
                        bb.debug(1, ...) messages are printed to stdout; -DD
                        sets the debug level to 2, where both bb.debug(1, ...)
                        and bb.debug(2, ...) messages are printed; etc.
                        Without -D, no debug messages are printed. Note that
                        -D only affects output to stdout. All debug messages
                        are written to ${T}/log.do_taskname, regardless of the
                        debug level.
  -q, --quiet           Output less log message data to the terminal. You can
                        specify this more than once.
  -n, --dry-run         Don't execute, just go through the motions.
  -S SIGNATURE_HANDLER, --dump-signatures=SIGNATURE_HANDLER
                        Dump out the signature construction information, with
                        no task execution. The SIGNATURE_HANDLER parameter is
                        passed to the handler. Two common values are none and
                        printdiff but the handler may define more/less. none
                        means only dump the signature, printdiff means compare
                        the dumped signature with the cached one.
  -p, --parse-only      Quit after parsing the BB recipes.
  -s, --show-versions   Show current and preferred versions of all recipes.
  -e, --environment     Show the global or per-recipe environment complete
                        with information about where variables were
                        set/changed.
  -g, --graphviz        Save dependency tree information for the specified
                        targets in the dot syntax.
  -I EXTRA_ASSUME_PROVIDED, --ignore-deps=EXTRA_ASSUME_PROVIDED
                        Assume these dependencies don't exist and are already
                        provided (equivalent to ASSUME_PROVIDED). Useful to
                        make dependency graphs more appealing
  -l DEBUG_DOMAINS, --log-domains=DEBUG_DOMAINS
                        Show debug logging for the specified logging domains
  -P, --profile         Profile the command and save reports.
  -u UI, --ui=UI        The user interface to use (knotty, ncurses or taskexp
                        - default knotty).
  --token=XMLRPCTOKEN   Specify the connection token to be used when
                        connecting to a remote server.
  --revisions-changed   Set the exit code depending on whether upstream
                        floating revisions have changed or not.
  --server-only         Run bitbake without a UI, only starting a server
                        (cooker) process.
  -B BIND, --bind=BIND  The name/address for the bitbake xmlrpc server to bind
                        to.
  -T SERVER_TIMEOUT, --idle-timeout=SERVER_TIMEOUT
                        Set timeout to unload bitbake server due to
                        inactivity, set to -1 means no unload, default:
                        Environment variable BB_SERVER_TIMEOUT.
  --no-setscene         Do not run any setscene tasks. sstate will be ignored
                        and everything needed, built.
  --setscene-only       Only run setscene tasks, don't run any real tasks.
  --remote-server=REMOTE_SERVER
                        Connect to the specified server.
  -m, --kill-server     Terminate any running bitbake server.
```

```
--observe-only        Connect to a server as an observing-only client.
--status-only         Check the status of the remote bitbake server.
-w WRITEEVENTLOG, --write-log=WRITEEVENTLOG
                      Writes the event log of the build to a bitbake event
                      json file. Use '' (empty string) to assign the name
                      automatically.
--runall=RUNALL       Run the specified task for any recipe in the taskgraph
                      of the specified target (even if it wouldn't otherwise
                      have run).
--runonly=RUNONLY     Run only the specified task within the taskgraph of
                      the specified targets (and any task dependencies those
                      tasks may have).
```

1.5.2. Примеры

1.5.2.1. Выполнение задач для одного задания

Выполнение задачи для одного задания сравнительно просто. BitBake анализирует указанный файл и выполняет задачу. Если задача не указана, BitBake выполняет принятую по умолчанию задачу build. BitBake всегда соблюдает зависимости между задачами.

Например, команда bitbake -b foo_1.0.bb будет выполнять задачу build для foo_1.0.bb, а команда bitbake -b foo.bb -c clean - задачу clean для foo_1.0.bb. Опция -b явно отменяет обработку зависимостей для задания.

1.5.2.2. Выполнение задач для набора заданий

Управление несколькими файлами .bb вызывает ряд сложностей. Ясно, что должен быть способ указать BitBake, какие файлы доступны и какие нужно выполнить. Кроме того, каждое задание должно указать свои зависимости как при сборке, так и при работе. Должен быть способ указать выбора заданий при обеспечении одних функций разными заданиями или при наличии нескольких версий задания.

Команда без опций --buildfile и -b воспринимает лишь PROVIDES и нет возможности предоставить что-либо иное. По умолчанию файл задания обычно указывает в PROVIDES своё имя пакета (packagename), как показано ниже.

```
$ bitbake foo
```

В следующем примере PROVIDES содержит имя пакета и используется опция -c, заставляющая BitBake выполнить задачу do_clean.

```
$ bitbake -c clean foo
```

1.5.2.3. Выполнение списка задач и заданий

Команды BitBake поддерживают указание разных задач для отдельных целей при задании нескольких целей. Например, имеется две цели (или задания) myfirstrecipe и mysecondrecipe и нужно указать BitBake задачу taskA для первого и taskB для второго. Это можно сделать командой вида

```
$ bitbake myfirstrecipe:do_taskA mysecondrecipe:do_taskB
```

1.5.2.4. Построение графа зависимостей

BitBake может создавать графы зависимостей, используя синтаксис с точками. Граф можно преобразовать в изображение с помощью Graphviz.

При создании графа зависимостей BitBake записывает в текущий рабочий каталог 3 файла:

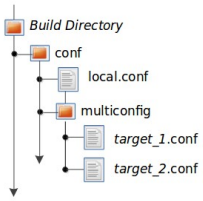
- *recipe-depends.dot* показывает зависимости между заданиями (сжатый вариант task-depends.dot);
- *task-depends.dot* показывает зависимости между задачами, соответствующие внутреннему списку выполнения;
- *pn-buildlist* показывает простой список целей, для которых выполняется сборка.

Для отключения базовых зависимостей служит опция -I и BitBake исключает эти зависимости из графа, что может сделать граф более читаемым. Таким образом можно удалить из графа DEPENDS унаследованных классов, например, base.bbclass. Ниже приведены два примера построения графа и во втором пропускаются базовые зависимости OE.

```
$ bitbake -g foo
$ bitbake -g -I virtual/kernel -I eglibc foo
```

1.5.2.5. Сборка с несколькими конфигурациями

BitBake может собирать в одной команде множество образов или пакетов, где разным целям нужны разные конфигурации. В таком сценарии каждую цель обозначают multiconfig. Для сборки с множеством конфигураций нужно задать конфигурацию отдельно для каждой цели с использованием файла параллельной конфигурации в каталоге сборки. Файлы конфигураций multiconfig должны размещаться внутри текущего каталога сборки в подкаталоге conf/multiconfig, как показано на рисунке для двух целей.



Причина такой иерархии заключается в том, что переменная BBPATH не создаётся до анализа уровней. Поэтому использование конфигурационного файла в качестве заданной заранее конфигурации невозможно, пока он не размещён в текущем рабочем каталоге.

Каждый файл конфигурации должен определять по меньшей мере машину и временный каталог, используемый BitBake для сборки. Лучше применять не пересекающиеся каталоги сборки.

Помимо файлов конфигурации для каждой цели, нужно также разрешить BitBake выполнять сборку для множества конфигураций. Это делается путём установки переменной BBMULTICONFIG в файле local.conf. В качестве примера рассмотрим конфигурационные файлы для целей target1 и target2, определённые в каталоге сборки. Приведённая ниже строка в файле local.conf позволит BitBake выполнить сборку для двух multiconfig.

```
BBMULTICONFIG = "target1 target2"
```

Когда конфигурационные файлы для целей созданы и BitBake разрешена сборка для нескольких конфигураций, применяется команда вида

```
$ bitbake [multiconfig:multiconfigname:]target [[[multiconfig:multiconfigname:]target] ... ]
```

Для помянутого выше примера с target1 и target2 команда будет иметь форму

```
$ bitbake multiconfig:target1:target multiconfig:target2:target
```

1.5.2.6. Включение зависимостей при сборке с множеством конфигураций

Иногда при сборке с множеством конфигураций могут существовать зависимости между целями (multiconfig). Например, при сборке образа для определённой архитектуры может потребоваться корневая файловая система для иной архитектуры. Иными словами, образ для первой конфигурации multiconfig зависит от корневой файловой системы второй multiconfig. Эта зависимость по существу заключается в том, что задача в задании для одной конфигурации multiconfig зависит от выполнения задания для другой конфигурации multiconfig. Для включения зависимостей при сборке с несколькими конфигурациями нужно объявить зависимости в задании, как показано ниже.

```
task_or_package[mcdepends] = "multiconfig:from_multiconfig:to_multiconfig:recipe_name:task_on_which_to_depend"
```

Для более детальной иллюстрации рассмотрим пример в 2 multiconfig - target1 и target2.

```
image_task[mcdepends] = "multiconfig:target1:target2:image2:rootfs_task"
```

Здесь from_multiconfig - это target1, а to_multiconfig — target2. Задача, для которой образ с заданием содержит image_task, зависит от завершения задачи rootfs_task, используемой для сборки image2, связанного с target2.

После установки зависимости можно собрать target1, используя команду

```
$ bitbake multiconfig:target1:image1
```

Эта команда выполняет все задачи, нужные для создания image1 в target1 multiconfig. Зависимость заставляет BitBake выполнить также задачу rootfs_task для сборки target2 multiconfig.

Зависимость задания от корневой файловой системы другой сборки может показаться не очень полезной. Рассмотрим изменённый вариант задания image1.

```
image_task[mcdepends] = "multiconfig:target1:target2:image2:image_task"
```

В этом случае BitBake нужно создать image2 для сборки target2, от которой зависит сборка target1. Поскольку target1 и target2 включены для сборок с несколькими конфигурациями и имеют разные конфигурационные файлы, BitBake помещает результаты каждой сборки в свой временный сборочный каталог.

Глава 2. Выполнение

Основной целью запуска BitBake является тот или иной вывод (установочный пакет, ядро, SDK или даже полный загружаемый образ Linux с ядром, загрузчиком и корневой файловой системой). Можно запускать bitbake с опциями для выполнения одной задачи, сборки одного задания, извлечения или очистки данных или получения данных о среде.

В этой главе полностью рассматривается процесс работы BitBake от старта до завершения. Для запуска служит команда вида

```
$ bitbake target
```

Информация о командах и опциях BitBake приведена в разделе 1.5. Команда bitbake.

Перед выполнением BitBake следует включить параллельную работу на хосте сборки с помощью переменной BB_NUMBER_THREADS в конфигурационном файле проекта local.conf. Для определения значения переменной можно воспользоваться командой grep processor /proc/cpuinfo, которая покажет число процессорных ядер хоста. В некоторых дистрибутивах Linux (например, Debian и Ubuntu) задачу можно решить с помощью команды ncpus.

2.1. Анализ базовых метаданных

Первым делом BitBake выполняет синтаксический анализ данных базовой конфигурации, хранящихся в файле bblayers.conf вашего проекта, который указывает BitBake нужные уровни, файлы layer.conf (1 на каждом уровне) и bitbake.conf. Сами данные включают три различных типа:

- *задания* содержат детали отдельных программных компонент;
- *данные классов* - абстракция базовых данных сборки (например, как собирать ядро Linux)№
- *данные конфигурации* - машинозависимые настройки, правила и т. п., собирающие конфигурацию воедино.

Файлы layer.conf служат для создания таких переменных, как BBPATH и BBFILES. Первая служит для поиска файлов конфигурации и классов в каталогах conf и classes, а вторая - для поиска файлов заданий и дополнений (.bb и .bbappend). При отсутствии файла bblayers.conf предполагается, что пользователь установил переменные BBPATH и BBFILES непосредственно в рабочей среде.

Далее с помощью созданной переменной BBPATH отыскивается файл bitbake.conf, который может включать дополнительные файлы с помощью директив include и require.

Перед анализом конфигурационных файлов Bitbake просматривает ряд переменных, включая:

- BB_ENV_WHITELIST;
- BB_ENV_EXTRAWHITE;
- BB_PRESERVE_ENV;

- BB_ORIGENV;
- BITBAKE_UI.

Первые 4 переменных задают отношение BitBake к переменным среды в процессе работы. По умолчанию BitBake очищает эти переменные и использует свои настройки, однако первая переменная позволяет указать переменные среды, сохраняемые BitBake (см. параграф 3.6.3. Передача информации в среду сборки задачи).

Метаданные базовой конфигурации являются глобальными и влияют на все выполняемые задачи и задания.

BitBake сначала просматривает текущий рабочий каталог в поиске conf/bblayers.conf, где предполагается переменная BBLAYERS с разделенным пробелами списков каталогов уровней. Если BitBake не находит файл bblayers.conf, предполагается установка пользователем переменных BBPATH и BBFILES непосредственно в окружении. Для каждого каталога (уровня) из списка отыскивается и анализируется файл conf/layer.conf с установкой в переменной LAYERDIR имени каталога, содержащего уровень. Идея состоит в автоматической установке BBPATH и других переменных с учётом данного каталога сборки.

Затем BitBake ищет файл conf/bitbake.conf по заданной пользователем переменной BBPATH. Этот файл обычно использует директивы include для добавления других метаданных, относящихся к архитектуре, машине, локальному окружению и т. п.

В файлах .conf можно указывать лишь определения переменных и директивы include. Некоторые переменные напрямую влияют на поведение BitBake. Такие переменные могут быть установлены непосредственно из среды в зависимости от переменных окружения, упомянутых выше, и установок в файлах конфигурации.

После анализа конфигурационных файлов BitBake использует механизм наследования для некоторых стандартных классов, анализируя класс при наличии указывающей его директивы inherit. Класс base.bbclass включается всегда, как и другие классы, указанные в конфигурации переменной INHERIT. BitBake ищет файлы классов в каталоге classes по пути BBPATH как и конфигурационные файлы. Хорошим способом узнать о файлах конфигурации и классах, используемых в среде выполнения является команда bitbake -e > mybb.log. Файлы классов и конфигурации будут указаны в начале файла mybb.log.

Нужно понимать, как BitBake анализирует фигурные скобки. Если в задании используется закрывающая фигурная скобка внутри функции без предшествующих пробелов, анализ BitBake завершается ошибкой. При использовании пары фигурных скобок в функции оболочки закрывающая скобка не должна указываться в начале строки без предшествующих пробелов. Ниже приведен пример, при анализе которого BitBake выдаст ошибку.

```
fakeroot create_shar() {
     cat << "EOF" > ${SDK_DEPLOY}/${TOOLCHAIN_OUTPUTNAME}.sh
usage()
{
  echo "test"
  ###### The following "}" at the start of the line causes a parsing error ######
}
EOF
}
```

Приведённый ниже вариант ошибки не вызовет.

```
fakeroot create_shar() {
    cat << "EOF" > ${SDK_DEPLOY}/${TOOLCHAIN_OUTPUTNAME}.sh
usage()
{
  echo "test"
  #####The following "}" with a leading space at the start of the line avoids the error #####
 }
EOF
}
```

2.2. Нахождение и анализ заданий

В фазе настройки BitBake устанавливает переменную BBFILES и использует её для создания списка заданий вместе с файлами дополнения (.bbappend) для выполнения. BBFILES содержит разделённый пробелами список файлов, в именах которых можно использовать шаблоны, как показано ниже.

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

BitBake анализирует каждое задание и дополнение из переменной BBFILES и записывает значения переменных в хранилище. Файлы дополнения применяются в порядке их следования в BBFILES.

Для каждого файла создаётся свежая копия базовой конфигурации, после чего задание анализируется построчно. Для каждого оператора inherit BitBake находит и анализирует файл класса (.bbclass), используя для поиска BBPATH. В заключение BitBake анализирует по порядку все файлы дополнения, найденные в BBFILES. Применяется базовое соглашение - использовать имя задания для определения частей метаданных. Например, в bitbake.conf имя и версия задания могут использоваться для определения переменных PN и PV, как показано ниже.

```
PN = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"
PV = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[1] or '1.0'}"
```

В этом примере задание something_1.2.3.bb установит для PN значение something, а для PV - 1.2.3.

К моменту завершения разбора задания BitBake получает список определённых задач и набор данных, состоящих из ключей и значений, а также сведения о зависимостях между задачами. Не вся эта информация требуется BitBake и

реально применяется лишь небольшая её часть для принятия решений о задании. Поэтому BitBake кэширует нужные значения, забывая остальное. Опыт показывает, что быстрее снова проанализировать метаданные, чем записывать и потом загружать их.

По возможности, последующие команды BitBake применяют кэшированные данные задания. Пригодность кэша определяется сначала расчётом контрольной суммы данных базовой конфигурации (BB_HASHCONFIG_WHITELIST) и проверкой совпадения. Совпадение контрольных сумм в расчёте и кэше говорит о неизменности задания и файлов классов, позволяя Bitbake снова использовать их без повторного анализа.

Имеются наборы файлов заданий, позволяющие пользователю держать множество репозиториев файлов .bb для одного пакета. Например, можно их использовать для создания своей копии восходящего репозитория с изменениями, которые в этом репозитории не нужны.

```
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"
```

Здесь снова предпочтительным методом организации кода являются уровни. Хотя набор кода остаётся, его задача состоит в установке приоритета для уровней и устранении перекрытий (конфликтов) между ними.

2.3. Провайдеры

После получения команды и завершения анализа заданий BitBake начинает выяснять, как собрать цель, просматривая список PROVIDES для каждого задания, включающий имена, под которыми задание может быть известно. Список PROVIDES в каждом задании может задаваться неявно через переменную PN или явно через необязательную переменную PROVIDES. При использовании переменной PROVIDES функциональность задания может быть найдена с другим именем, нежели задано в PN. Например, задание keyboard_1.0.bb может включать строку PROVIDES += "fullkeyboard". Список PROVIDES для этого задания принимает значение keyboard (неявное) и fullkeyboard (явное). В результате функциональность keyboard_1.0.bb может быть найдена по двум именами.

2.4. Предпочтения

Список PROVIDES обеспечивает лишь часть решения для заданий цели. Поскольку у цели может быть множество провайдеров, BitBake нужно расставить приоритеты для выбора. Типичным примером цели с множеством провайдеров является virtual/kernel в списках PROVIDES каждого задания для ядра. Зачастую каждая машина выбирает лучшего поставщика ядра, указывая в своём конфигурационном файле строку вида

```
PREFERRED_PROVIDER_virtual/kernel = "linux-yocto"
```

Используемый по умолчанию PREFERRED_PROVIDER является поставщиком с указанным переменной именем. BitBake перебирает все нужные цели для построения и выполнения зависимостей.

Выбор провайдера осложняется возможностью наличия у данного поставщика множества версий. По умолчанию BitBake выбирает старшую версию, сравнивая номера как принято в Debian. Можно указать предпочтительную версию в переменной PREFERRED_VERSION, а порядок можно менять с помощью переменной DEFAULT_PREFERENCE.

По умолчанию файлы имеют уровень предпочтения 0. Установка DEFAULT_PREFERENCE = "-1" делает выбор задания маловероятным, если оно не указано явно. Установка DEFAULT_PREFERENCE = "1" делает использование задания вероятным. PREFERRED_VERSION переопределяет установку DEFAULT_PREFERENCE, которая часто применяется для маркировки новых и экспериментальных версий задания, пока оно не будет сочтено стабильным.

При наличии множества «версий» данного задания BitBake по умолчанию выбирает наиболее свежую, если не задано иное. Если в рассматриваемом задании установлено DEFAULT_PREFERENCE ниже, чем в других заданиях (по умолчанию 0), оно не будет выбрано. Это позволяет сопровождающим репозиторий задания указать свои предпочтения для выбираемой по умолчанию версии. Пользователь также может задать предпочтительную версию.

Если первое задание называется a_1.1.bb, переменная PN будет иметь значение a, PV - 1.1. Если имеется задание a_1.2.bb, BitBake по умолчанию выберет его. Однако можно указать в файле .conf, анализируемом BitBake, переменную PREFERRED_VERSION_a = "1.1".

Обычно в задании указывают две версии - стабильную с номером (предпочтительная) и автоматически извлекаемую из репозитория, которая считается «передовой», но может быть выбрана только явно. Например, в базе кодов OE имеется стандартное задание BusyBox с версией - busybox_1.22.1.bb, а также Git-версия busybox_git.bb, для которой указано DEFAULT_PREFERENCE = "-1", чтобы отдать предпочтение стабильной версии.

2.5. Зависимости

Сборка каждой цели в BitBake состоит из множества задач, включая выборку, распаковку, применение правок (patch), настройку и компиляцию. Для обеспечения высокой производительности на многоядерных системах BitBake рассматривает каждую задачу как независимую единицу со своим набором зависимостей, определяемых несколькими переменными (Глава 5. Глоссарий переменных). На базовом уровне достаточно разобраться с использованием переменных DEPENDS и RDEPENDS. Обработка зависимостей в BitBake рассмотрена в разделе 3.10. Зависимости.

2.6. Список задач

На основе созданного списка провайдеров и данных о зависимостях BitBake может точно рассчитать порядок выполнения задач (3.6. Задачи). Сборка начинается с того, что BitBake организует множество потоков, вплоть до значения BB_NUMBER_THREADS. Ветвление продолжается, пока есть задачи для запуска и не превышен предел. Следует отметить, что значение переменной BB_NUMBER_THREADS существенно влияет на производительность.

По завершении каждой задачи записывается временная мета в каталог, заданный переменной STAMP. При следующем запуске BitBake просматривает каталог сборки и не повторяет задачи с действительной временной меткой.

Действие метки определяется на уровне файла задания. Например, если штамп задачи configure имеет временную метку, превышающую метку для задачи compile, компиляция будет запущена снова, однако это не будет влиять на других провайдеров, зависящих от этой цели.

Формат штампов частично настраивается и в современных версиях BitBake к штампу добавляется хэш-значение, что позволяет автоматически делать непригодным прежний штамп при изменении конфигурации. Это хэш-значение (подпись) определяется выбранной политикой подписей (2.8. Контрольные суммы (подписи)). Можно также добавить к штампу дополнительные метаданные с помощью флага задачи [stamp-extra-info]. Например, OE применяет этот флаг для задания зависимости некоторых задач от машины. Некоторые задачи помечаются как nostamp и штампы для них не создаются, что приводит к перезапуску такой задачи всякий раз.

2.7. Выполнение задач

Задачи могут выполняться в оболочке (shell) или Python. Для shell-задач BitBake записывает сценарий в файл ${T}/run.do_taskname.pid и выполняет его. Созданный сценарий содержит все экспортируемые переменные и функции оболочки с преобразованными переменными. Вывод сценария записывается в файл ${T}/log.do_taskname.pid. Просмотр преобразованных shell-функций в файле run и вывода в файле log полезен при отладке.

Задачи Python выполняются внутри BitBake с выводом информации на терминал управления. В новых версиях BitBake будет записывать функции и вывод в файлы, как это делается для shell-задач.

Порядок выполнения задач контролируется планировщиком, который можно настроить для конкретных ситуаций через переменные BB_SCHEDULER и BB_SCHEDULERS. Функции могут выполняться до и после основной задачи. Это управляется с помощью флагов [prefuncs] и [postfuncs] в списке выполняемых задач.

2.8. Контрольные суммы (подписи)

Контрольная сумма является уникальной подписью ввода в задачу и может служить для решения вопроса о перезапуске задачи. Поскольку изменение ввода ведёт к перезапуску задач, BitBake нужно детектировать изменение ввода. Для shell-задач это достаточно просто, поскольку BitBake создает сценарии run для каждой задачи и можно рассчитать контрольную сумму, которая покажет изменение данных задачи. Однако в контрольную сумму включается не все. Во-первых, имеется фактический путь сборки задачи - рабочий каталог и его изменение не должно влиять на вывод целевых пакетов. Упрощённый вариант исключения рабочего каталога из контрольной суммы заключается в установке фиксированного значения. BitBake идёт дальше и применяет переменную BB_HASHBASE_WHITELIST для задания списка переменных, которые не следует учитывать в контрольных суммах.

Другая проблема связана с наличием в сценариях run функций, которые могут не вызываться. Решение для инкрементальной сборки содержит код, учитывающий зависимости между shell-функциями, который служит для урезания сценариев run до минимального набора функций, что существенно смягчает эту проблему и делает сценарии более читаемыми.

Такой же подход применяется к задачам Python. Процесс должен выяснить переменные, к которым обращается функция Python, и вызываемые ею функции. Решение для инкрементальной сборки содержит код, который сначала определяет зависимости переменных и функций, а затем создает контрольную сумму для входных данных задачи. Как и для рабочих каталогов, имеются случаи, где следует игнорировать зависимости. В таких ситуациях можно указать это процессу сборки в виде PACKAGE_ARCHS[vardepsexclude] = "MACHINE", где указано, что переменная PACKAGE_ARCHS не зависит от значения MACHINE, даже если она ссылается на неё. Существуют и случаи, когда нужно добавить зависимости, которые BitBake не может найти. Это можно сделать в форме PACKAGE_ARCHS[vardeps] = "MACHINE", где переменная MACHINE явно указана как зависимость для PACKAGE_ARCHS.

Возможен случай с Python, где BitBake не может определить зависимости. При работе в режиме отладки (-DDD), BitBake выдаёт сообщения при обнаружении невозможности выяснить зависимости.

До этого рассматривался лишь прямой ввод в задачи. Вводимая таким образом информация в коде называется базовым хэшем (basehash). Однако остаётся косвенный ввод - элементы, которые уже собраны и имеются в каталоге сборки. Контрольная сумма (подпись) для конкретной задачи должна учитывать хэш-значения всех задач, от которых она зависит. Выбор зависимостей для добавления задаёт политика и в результате создаётся основная контрольная сумма, объединяющая basehash и хэш-значения всех задач, от которых имеются зависимости.

На уровне кода есть много способов воздействия на хэш-значения. В файле конфигурации BitBake можно задать дополнительные данные для создания basehash. Приведённое ниже выражение из OE фактически исключает зависимости от глобальных переменных (т. е. они не включаются в контрольную сумму).

```
BB_HASHBASE_WHITELIST ?= "TMPDIR FILE PATH PWD BB_TASKHASH BBPATH DL_DIR \
    SSTATE_DIR THISDIR FILESEXTRAPATHS FILE_DIRNAME HOME LOGNAME SHELL TERM \
    USER FILESPATH STAGING_DIR_HOST STAGING_DIR_TARGET COREBASE PRSERV_HOST \
    PRSERV_DUMPDIR PRSERV_DUMPFILE PRSERV_LOCKDOWN PARALLEL_MAKE \
    CCACHE_DIR EXTERNAL_TOOLCHAIN CCACHE CCACHE_DISABLE LICENSE_PATH SDKPKGSUFFIX"
```

В примере не указан рабочий каталог, являющийся частью TMPDIR.

Правила включения хэш-значений с учётом цепочек зависимостей являются более сложными и обычно реализуются функциями Python. Код в файле meta/lib/oe/sstatesig.py содержит два примера т показывает, как можно включить свои правила в систему. Этот файл определяет два базовых генератора подписей, используемых в OpenEmbedded-Core, - OEBasic и OEBasicHash. По умолчанию в BitBake включён фиктивный обработчик подписей noop. Это означает, что поведение не отличается от предыдущих версий. OE-Core использует по умолчанию обработчик OEBasicHash (файл bitbake.conf)

```
BB_SIGNATURE_HANDLER ?= "OEBasicHash"
```

OEBasicHash в отличие от OEBasic добавляет хэш задачи в файлы штампов. Это ведёт к тому, что любое изменение метаданных, влияющее на хэш задачи, автоматически вызывает повторный запуск задачи. В результате не нужно увеличивать значения PR, а изменения метаданных автоматически распространяются по сборке.

Следует отметить, что конечным результатом генерации подписей является доступность в сборке некоторых данных о зависимостях и хэш-значениях, включая:

- *BB_BASEHASH_task-taskname* - базовые хэш-значения для каждой задачи в задании;
- *BB_BASEHASH_filename:taskname* - базовые хэш-значения для каждой зависимой задачи;
- *BBHASHDEPS_filename:taskname* - зависимости для каждой задачи;
- *BB_TASKHASH* - хэш текущей работающей задачи.

Отметим, что опция -S позволяет отлаживать подписи в BitBake, поддерживая различные режимы с использованием отладочных функций BitBake или указанного в метаданных/подписи обработчика. Простейшим параметром является none, обеспечивающий вывод информации о подписях в каталог STAMPS_DIR указанной цели. Указание параметра printdiff заставляет BitBake пытаться найти максимальное соответствие (например, в кэше sstate), а затем запустить bitbake-diffsigs для совпадений с целью определения штампа и места, где дерево штампов расходится. В новых версиях BitBake могут появиться другие обработчики подписей, запускаемые новыми параметрами опции -S. Информация о контрольных суммах метаданных приведена в параграфе 3.12. Контрольные суммы задач и Setscene.

2.9. Setscene

Процесс setscene позволяет BitBake обрабатывать ранее собранные элементы, не создавая их каждый раз с нуля. Для этого нужны надёжные сведения о совместимости с имеющимся элементом. Описанные выше подписи представляют идеальный вариант контроля совместимости и при совпадении подписей объект можно использовать снова. При повторном использовании объектов возникает проблема замены результата данной задачи или набора задач предварительно собранным элементом. Для решения проблемы в BitBake используется процесс setscene.

Когда у BitBake запрошена сборка данной цели, программа сначала проверяет доступность кэшированных данных для собираемых или промежуточных целей. При наличии такой информации BitBake использует её вместо запуска задач. Сначала BitBake вызывает функцию, определённую переменной BB_HASHCHECK_FUNCTION, со списком задач и соответствующих хэш-значений для планируемой сборки. Эта функция обеспечивает быстрый возврат списка задач, для которых могут быть получены собранные результаты. Затем для каждой возвращённой задачи BitBake выполняет setscene-версию, включающую возможный результат. Setscene-версии задач имеют суффикс _setscene. Эти задачи выполняются и предоставляют нужные элементы (возможно, возвращая вместо этого отказ).

Как было отмечено выше, элемент может охватывать несколько задач. Например, нет смысла получать компилятор при наличии скомпилированной версии. Для обработки таких ситуаций BitBake вызывает функцию BB_SETSCENE_DEPVALID для каждой успешной задачи setscene, чтобы решить вопрос о получении зависимостей этой задачи.

После выполнения всех задач setscene BitBake вызывает функцию, указанную в BB_SETSCENE_VERIFY_FUNCTION2, со списком задач, которые BitBake считает «охваченными». Метаданные могут гарантировать корректность списка и информировать BitBake о необходимости перезапуска определённой задачи, независимо от результата setscene.

Метаданные setscene описаны в параграфе 3.12. Контрольные суммы задач и Setscene.

Глава 3. Синтаксис и операторы

Файлы Bitbake используют свой синтаксис, который похож на синтаксис других языков, но имеет особенности.

3.1. Базовый синтаксис

3.1.1. Установка переменных

Для незамедлительной установки в переменной VARIABLE значения value используется выражение VARIABLE = "value". Такое назначение называется «жёстким» (hard). Если назначаемое значение включает пробел, он сохраняется.

```
VARIABLE = " value"
VARIABLE = "value "
```

Установка "" делает переменную пустой, а " " помещает в неё. пробел (это разные вещи).

```
VARIABLE = ""
VARIABLE = " "
```

Вместо двойных кавычек можно применять одинарные, что позволяет включать в значение двойные кавычки.

```
VARIABLE = 'I have a " in my value'
```

В отличие от оболочек Bourne одинарные кавычки эквивалентны двойным и не препятствуют преобразованию переменных.

3.1.2. Изменение имеющихся переменных

Иногда нужно изменить имеющиеся переменные, для чего существует несколько способов:

- отредактировать задание, включающее переменную;
- изменить принятое по умолчанию значение переменной в файле класса .bbclass;
- изменить переменную в файле дополнения .bbappend для переопределения её значения;
- изменить переменную в файле конфигурации для переопределения её значения.

Изменение значения переменной иногда может зависеть от способа его исходного назначения, а также от желаемого эффекта изменения. В частности, если добавить значение в конец заданного по умолчанию значения переменной, результат может оказаться не соответствующим ожиданиям. Если изменение даёт неожиданный результат, можно проверить действительное значение с помощью BitBake как для конфигурации, так и для задания.

- Для проверки конфигурационных изменений служит команда bitbake -e, которая выводит значения переменных после анализа файлов конфигурации (local.conf, bblayers.conf, bitbake.conf и т. п.). Экспортируемые в среду переменные имеют префикс export в выводе команды.
- Для проверки изменений в задании служит команда bitbake recipe -e | grep VARIABLE=", которая позволяет проверить реальное значение переменной VARIABLE.

3.1.3. Объединение строк

Вне функций BitBake соединяет любую строку, заканчивающуюся символом \, со следующей строкой до начала синтаксического анализа. Символ \ обычно применяется в многострочном присваивании значений, как показано ниже.

```
FOO = "bar \
       baz \
       qaz"
```

При объединении символ \ и символы новой строки удаляются и строка FOO получит значение "bar baz qaz". Ниже приведены два варианта присвоения значения barbaz переменной FOO.

```
FOO = "barbaz"
FOO = "bar\
baz"
```

BitBake не интерпретирует последовательности типа \n в значениях переменных. Для специальной трактовки их следует передавать таким утилитам как printf или echo -n.

3.1.4. Преобразование переменных

Переменные могут указывать содержимое других переменных с использованием синтаксиса похожего на преобразование переменных в оболочках Bourne. Приведённые ниже назначения обеспечивают в переменной B значение preavalpost.

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

В отличие от Bourne фигурные скобки обязательны и преобразование выполняется лишь для ${FOO}, но не для $FOO.

Оператор = не ведёт к незамедлительному преобразованию переменной справа, оно откладывается до получения этой переменной значения, которое действительно используется. Таким образом, результат зависит от текущего значения указанных переменных, как показано ниже.

```
A = "${B} baz"
B = "${C} bar"
C = "foo"
```

В этот момент ${A} имеет значение "foo bar baz".

```
C = "qux"
```

В этот момент ${A} имеет значение "qux bar baz".

```
B = "norf"
```

В этот момент ${A} имеет значение "norf baz".

Другое поведение обеспечивается оператором незамедлительного преобразования :=.

Если синтаксис преобразования использован для несуществующей переменной, назначение сохраняется буквально. Например, при указании BAR = "${FOO}" строка BAR получит значение ${FOO}, если переменной FOO не существует.

3.1.5. Установка с сохранением принятого по умолчанию значения (?=)

Можно использовать оператор ?= для «более мягкого» назначения переменной. Этот оператор позволяет определить переменную, если на момент анализа она ещё не была определена, но сохраняет значение, если переменная уже задана. Например, при анализе выражения A ?= "aval" будет сохранено имеющееся значение переменной или задано значение aval, если переменная ещё не была назначена. Это присвоение выполняется сразу же, поэтому при наличии нескольких операторов ?= для одной переменной работать будет лишь первое.

3.1.6. Отложенная установка с сохранением заданной по умолчанию (??=)

Можно использовать более «слабый» вариант описанного в предыдущем параграфе оператора - ??=, который похож на ?=, но присвоение выполняется не сразу, а в конце процесса анализа. Поэтому при наличии нескольких операторов ??= действовать будет последний. Любые назначения = или ?= будут переопределять ??=. Например,

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

Если переменная A была установлена до анализа этих операторов, она сохранит значение, в противном случае получит значение someothervalue.

3.1.7. Незамедлительное преобразование переменной (:=)

Оператор := ведёт к незамедлительному преобразованию переменных без ожидания их реального использования.

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

В этом примере A получит значение "test 123" поскольку ${B} и ${A} в момент анализа не определены, а переменная C - "cvalappend", поскольку ${C} сразу же преобразуется в "cval".

3.1.8. Добавление в конец (+=) и в начало (=+) с пробелом

Добавление значения в конец или в начало переменной обеспечивается операторами += и =+, которые помещают пробел между имеющимся и добавленным значением. Эти назначения выполняются сразу же при анализе. Например,

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

Переменная B получит значение "bval additionaldata" а C - "test cval".

3.1.9. Добавление в (.=) и в начало (=.) без пробела

Для добавления в конец или начало переменной значения без пробела служат операторы .= и =., которые выполняются сразу же при анализе.

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

Переменная B получит значение "bvaladditionaldata", а C - "testcval".

3.1.10. Добавление в начало или в конец с переопределением

Можно также добавлять в начало или в конец значения переменной с использованием синтаксиса стиля переопределения. В этом случае пробелы не добавляются автоматически. Эти операторы отличаются от :=, .=, =., += и =+ в том, что они применяются по завершении синтаксического анализа, а не сразу же. Примеры приведены ниже.

```
B = "bval"
B_append = " additional data"
C = "cval"
C_prepend = "additional data "
D = "dval"
D_append = "additional data"
```

Переменная B в результате становится "bval additional data", C - "additional data cval", а D - "dvaladditional data".

При использовании этого синтаксиса следует самостоятельно контролировать добавление пробелов. Можно также применять операторы append и prepend к функциям оболочки и функциям Python в стиле BitBake. Примеры приведены в параграфах 3.5.1. Функции оболочки и 3.5.2. Функции Python в стиле BitBake.

3.1.11. Удаление

Можно удалять значения из списков с помощью синтаксиса переопределения. Указание удаляемого значения ведёт к исключению из переменной всех его вхождений. При использовании этого синтаксиса BitBake предполагает указание одной или нескольких строк. Например,

```
FOO = "123 456 789 123456 123 456 123 456"
FOO_remove = "123"
FOO_remove = "456"
FOO2 = "abc def ghi abcdef abc def abc def"
FOO2_remove = "abc def"
```

Переменная FOO в результате станет " 789 123456 ", FOO2 - " ghi abcdef ". Подобно _append и _prepend, оператор _remove применяется по завершении синтаксического анализа.

3.1.12. Преимущества стиля переопределения

Преимущество операторов стиля переопределения _append, _prepend и _remove по сравнению с += и =+ заключается в том, что они обеспечивают гарантированные операции. Рассмотрим в качестве примера класс foo.bbclass, который должен добавить значение val к переменной FOO, и задание, использующее foo.bbclass, как показано ниже.

```
inherit foo
FOO = "initial"
```

Если foo.bbclass использует оператор +=, как показано ниже, значением FOO будет "initial", что не устраивает.

```
FOO += "val"
```

Если же foo.bbclass применяет оператор _append, переменная FOO получит нужно значение "initial val".

```
FOO_append = " val"
```

Никогда не следует применять += вместе с _append. Приведённые ниже операторы добавляют "barbaz" в конец FOO.

```
FOO_append = "bar"
FOO_append = "baz"
```

Если во втором выражении применить оператор +=, это добавить в окончательный результат пробел перед "baz".

Другое преимущество стиля переопределения заключается в том, что операторы можно комбинировать с другими переопределениями, как описано в разделе 3.3. Синтаксис условий.

3.1.13. Синтаксис флагов переменных

Флаги переменных в BitBake служат реализацией свойств или атрибутов переменных, позволяя связывать с ними дополнительную информацию. Описание флагов приведено в разделе 3.7. Флаги переменных.

Флаги можно определять и добавлять в начало или в конец имеющихся. Все описанные выше стандартные операции, кроме синтаксиса стиля переопределения, применимы к флагам (например, _prepend, _append и _remove). Например,

```
FOO[a] = "abc"
FOO[b] = "123"
FOO[a] += "456"
```

Переменная FOO имеет флаги [a] и [b], для которых сразу же устанавливаются значения abc и 123. Затем флаг [a] становится "abc 456". Нет нужды в создании предопределённых флагов переменных, поскольку начать их использование можно в любой момент. Одним из наиболее распространённых применений флагов является краткое описание переменных BitBake в форме CACHE[doc] = "The directory holding the cache of the metadata.".

3.1.14. Преобразование переменных Python

Можно использовать встроенное в Python преобразование переменных для установки значений. Например,

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

Для переменной DATE будет установлено текущее время.

Возможно наиболее частым применением этой функции является извлечение значений переменных из внутреннего словаря BitBake (d). Приведённые ниже строки извлекают имя и версию пакета.

```
PN = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"
PV = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[1] or '1.0'}"
```

Inline-выражения Python работают как преобразования переменных с операторами = и :=. Например, при назначении FOO = "${@foo()}" функция foo() будет вызываться при каждом преобразовании FOO. При назначении FOO := "${@foo()}" функция foo() будет вызываться лишь однократно при анализе назначения. Другой способ установки переменных в коде Python при синтаксическом анализе описан в параграфе 3.5.5. Анонимные функции Python.

3.1.15. Отмена переменных

Можно полностью удалить переменную или флаг из внутреннего словаря данных BitBake с помощью оператора unset.

```
unset DATE
unset do_fetch[noexec]
```

3.1.16. Указание путей

При задании путей для использования в BitBake символ ~ не применяется в качестве замены домашнего каталога, поскольку BitBake не преобразует этот символ, как это делают командные процессоры. Нужно указывать полный путь.

```
BBLAYERS ?= " \
  /home/scott-lenovo/LayerA \
  "
```

3.2. Экспорт переменных в среду

Можно экспортировать переменные в среду выполняемых задач с помощью оператора export. Например, задача do_foo будет при работе печатать "value from the environment", если задать

```
export ENV_VARIABLE
ENV_VARIABLE = "value from the environment"

do_foo() {
    bbplain "$ENV_VARIABLE"
}
```

BitBake не преобразует в этом случае переменную $ENV_VARIABLE по причине отсутствия фигурных скобок и переменная преобразуется оболочкой. Размещение строки export ENV_VARIABLE относительно присвоения значения переменной не играет роли. Можно объединить назначение и экспорт в форме export ENV_VARIABLE = "variable-value".

В выводе команды bitbake -e экспортируемые в среду переменные помечаются префиксом export.

Обычно экспортируемые переменные включают CC и CFLAGS, которые принимают многие системы сборки.

3.3. Синтаксис условий

BitBake использует переменную OVERRIDES для указания переменных, которые переопределяются после анализа задания и файлов конфигурации. Здесь описано применение OVERRIDES в качестве условных метаданных, преобразование ключей применительно к OVERRIDES, а также приведены примеры использования.

3.3.1. Условные метаданные

Можно применять OVERRIDES в качестве условия для выбора конкретной версии переменной и условного добавления значения в начало или в конец. В переопределениях можно использовать только символы нижнего регистра, символы подчёркивания допускаются в именах переопределений только в качестве разделителей имён.

- *Выбор переменной.* OVERRIDES содержит список разделенных двоеточиями элементов, для которых нужно выполнить условия. Например, если есть условная переменная для arm и arm входит в OVERRIDES, будет использована связанная с arm версия переменной вместо безусловной. Например,

  ```
  OVERRIDES = "architecture:os:machine"
  TEST = "default"
  TEST_os = "osspecific"
  TEST_nooverride = "othercondvalue"
  ```

  OVERRIDES содержит 3 переопределения - architecture, os и machine. Переменная TEST по умолчанию имеет значение default. Можно выбрать связанную с os переменную TEST, добавив в конец переопределение os (TEST_os).

  Для лучшего понимания рассмотрим практический пример, предполагающий файл задания для ядра Linux в OE на основе метаданных. Приведённые ниже строки из файла задания сначала устанавливают в переменной ветви ядра KBRANCH принятое по умолчанию значение, затем по условию переопределяют его в соответствии с архитектурой.

  ```
  KBRANCH = "standard/base"
  KBRANCH_qemuarm  = "standard/arm-versatile-926ejs"
  ```

```
KBRANCH_qemumips = "standard/mti-malta32"
KBRANCH_qemuppc  = "standard/qemuppc"
KBRANCH_qemux86  = "standard/common-pc/base"
KBRANCH_qemux86-64  = "standard/common-pc-64/base"
KBRANCH_qemumips64 = "standard/mti-malta64"
```

- *Добавление в начало или в конец.* BitBake также поддерживает операции добавления в конец и в начало переменной в зависимости от наличия элемента в OVERRIDES. Например,

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

DEPENDS в результате принимает значение "glibc ncurses libmad". В примере с заданием для ядра можно использовать условное добавление в конце переменной в зависимости от архитектуры, как показано ниже.

```
KERNEL_FEATURES_append = " ${KERNEL_EXTRA_FEATURES}"
KERNEL_FEATURES_append_qemux86=" cfg/sound.scc cfg/paravirt_kvm.scc"
KERNEL_FEATURES_append_qemux86-64=" cfg/sound.scc cfg/paravirt_kvm.scc"
```

- *Установка для одной задачи.* BitBake поддерживает установку переменной лишь для выполнения 1 задачи.

```
FOO_task-configure = "val 1"
FOO_task-compile = "val 2"
```

Здесь переменная FOO имеет значение "val 1" при выполнении задачи do_configure и "val 2" - в do_compile.

Это реализуется путём добавления задачи (например, task-compile:) в начало OVERRIDES для локального хранилища задачи do_compile. Можно также использовать этот синтаксис с другими комбинациями (например, _prepend), как показано ниже.

```
EXTRA_OEMAKE_prepend_task-compile = "${PARALLEL_MAKE} "
```

3.3.2. Преобразование ключей

Ключи преобразуются, когда хранилище BitBake финализируется перед преобразованием переопределений.

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

В этом случае по завершении анализа но перед обработкой переопределений BitBake преобразует ${B} в 2. Это преобразование меняет значение A2, которое было Y, на X.

3.3.3. Примеры

Несмотря на приведённые выше разъяснения разных форм определения переменных, может оказаться сложно точно понять, что происходит при объединении операторов переменных с условными и безусловными переопределениями. В этом параграфе приведены некоторые примеры для таких преобразования, позволяющие разобраться лучше.

Часто возникает путаница с определением порядка применения переопределений и операторов append. Напомним, что операции _append и _prepend не меняют значения сразу, как это делают операторы =+, =., += и .=.

```
OVERRIDES = "foo"
A = "Z"
A_foo_append = "X"
```

В примере выше A безусловно получает значение Z, а X безусловно и незамедлительно добавляется в конец переменной A_foo. Поскольку переопределения ещё не было, к A_foo добавляется значение X, а A становится Z. Однако переопределение меняет картину. Поскольку переменная foo указана в OVERRIDES, условная переменная A заменяется версией foo, которая равна X. В результате A_foo заменяет A.

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

Здесь перед переопределением в переменной A устанавливается значение Z и A_append_foo становится X. После переопределения foo в конце A добавляется X и A становится ZX. Отметим, что пробел не добавляется.

В следующем примере используется обратный порядок добавления и переопределения.

```
OVERRIDES = "foo"
A = "Y"
A_foo_append = "Z"
A_foo_append = "X"
```

Здесь до переопределения A получает значение Y, затем в A_foo устанавливается значение Z, после чего в конце добавляется X и получается ZX. Затем переопределение foo приводит к тому, что условная переменная A получает значение ZX (A заменяется на A_foo).

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

В этом примере тип операторов append влияет на порядок назначений, поскольку BitBake проходит код несколько раз. Изначально в A устанавливается значение "1 45", поскольку три оператора используют незамедлительное назначение. Затем BitBake применяет операторы _append и A становится "1 4523".

3.4. Общая функциональность

BitBake обеспечивает возможность совместного использования метаданных через включаемые файлы (.inc) и классы (.bbclass). Например, можно определить задачу, которая будет применяться в нескольких заданиях. Для этого создаётся файл класса .bbclass с общей функциональностью и используется директива inherit в заданиях, которым нужна такая задача.

Ниже описаны механизмы BitBake для совместного использования функций - include, inherit, INHERIT и require.

3.4.1. Поиск включаемых файлов и классов
BitBake использует переменную BBPATH для поиска включаемых файлов и классов, просматривая также текущий каталог для директив include и require. Переменная BBPATH похожа на переменную среды PATH. Для того, чтобы включаемые файлы и классы были найдены BitBake, их нужно разместить в каталоге classes, указанном в BBPATH.

3.4.2. Директива inherit
При создании файла задания или класса можно использовать директиву inherit для наследования функциональности класса (.bbclass). BitBake разрешает применять эти директивы только в файлах заданий и классов (.bb и .bbclass).

Директива inherit является элементарным способом указания требуемой заданию функциональности, содержащейся в классе. Например, можно абстрагировать задачи сборки пакета с использованием Autoconf и Automake, поместив их в файл класса, а затем наследовать этот класс в файлах заданий. Например, задания могут использовать директиву inherit autotools для наследования файла autotools.bbclass. В таком случае BitBake будет искать файл classes/autotools.bbclass в BBPATH.

Можно переопределить любые значения и функции наследуемого класса внутри задания после оператора inherit.

Если нужно унаследовать несколько классов, они указываются в директиве inherit через пробелы. Например inherit buildhistory rm_work задаёт наследование классов buildhistory и rm_work. Преимуществом inherit перед директивами include и require является возможность условного наследования в форме inherit ${VARNAME}. Переменная VARNAME в таких случаях должна быть назначена до оператора inherit. Один из вариантов условного наследования имеет вид

```
VARIABLE = ""
VARIABLE_someoverride = "myclass"
```
Другим вариантом является использование анонимной функции Python. Например,

```
python () {
    if condition == value:
        d.setVar('VARIABLE', 'myclass')
    else:
        d.setVar('VARIABLE', '')
}
```
Можно также применять in-line-выражения Python в форме

```
inherit ${@'classname' if condition else ''}
inherit ${@functionname(params)}
```
В любом случае преобразование в пустую строку ошибки не вызывает, просто наследования не произойдёт.

3.4.3. Директива include
Директива include заставляет BitBake анализировать заданный файл и помещать его содержимое в указанное место. Это похоже на одноимённую директиву Make, но путь указывается относительный и BitBake включает первый файл, найденный по переменной BBPATH. Директива include обеспечивает более общий по сравнению с inherit способ включения, поскольку наследование ограничено файлами классов (.bbclass). Директива include применима к любому виду общей или инкапсулированной функциональности или конфигурации, которая не подходит для классов (.bbclass).

Например, можно включить в задание те или иные элементы самопроверки в форме include test_defs.inc.

Если включается директивой include файл не найден, ошибки не возникает. Если же нужно обязательно включить файл, следует применять директиву require.

3.4.4. Директива require
Директива require похожа на include, но BitBake возвращает ошибку, если файл не найден. Директива require, как и include, является более общим способом включения функциональности нежели inherit, которая работает только для классов (.bbclass). Директива применима для любой общей или инкапсулированной функциональности и конфигурации, которая не подходит для .bbclass. Подобно обработке include, при указании относительного пути в директиве require BitBake использует первый файл, найденный в BBPATH.

Предположим, что имеется 2 задания (например, foo_1.2.2.bb и foo_2.0.0.bb), где каждая версия включает некие общие функции. Можно создать включаемый файл foo.inc с общими определениями для сборки foo. Файл foo.inc должен размещаться в одном каталоге с файлами заданий. После это можно указать в заданиях require foo.inc.

3.4.5. Конфигурационная директива INHERIT
При создании конфигурационных файлов (.conf) можно применять директиву INHERIT для наследования классов. Например, можно добавить в конфигурацию наследованием файла abc.bbclass в форме INHERIT += "abc". Эта директива включает наследование указанного класса в точек анализа директивы. Файл .bbclass должен размещаться в каталоге classes одного из каталогов BBPATH. Поскольку файлы .conf анализируются первыми при работе BitBake, использование директивы INHERIT обеспечивает глобальное (для всех заданий) наследование класса. Если нужно наследовать несколько классов, они указываются через пробелы в форме INHERIT += "autotools pkgconfig".

3.5. Функции
Как в большинстве языков, функции являются основными блоками задач. В BitBake имеется несколько типов функций.

- *Функции оболочки*, написанные на языке сценариев, выполняются как функции и/или задачи.
- *Функции Python в стиле BitBake* на языке Python, выполняемые BitBake или другими функциями Python с помощью bb.build.exec_func().
- *Функции Python*, написанные на языке Python и выполняемые Python.

- *Анонимные функции Python*, функции Python, автоматически выполняемые в процессе анализа.

В файлах классов (.bbclass) и заданий (.bb или .inc) можно определять любые функции.

3.5.1. Функции оболочки

Функции оболочки или shell-функции используют язык сценариев оболочки и выполняются как задачи и/или функции. Их могут также вызывать другие shell-функции.

```
some_function () {
    echo "Hello World"
}
```

При создании таких функций в классе или задании нужно следовать правилам для программ оболочки. Сценарии выполняются оболочкой /bin/sh, в качестве которой не обязательно служит shell (может быть, например, dash), поэтому не следует применять специфичные для Bash сценарии.

В этих функциях можно применять переопределения и операторы с переопределением. Обычно это применяется в файлах .bbappend для изменения функций основного задания или унаследованных функций класса. Например,

```
do_foo() {
    bbplain first
    fn
}

fn_prepend() {
    bbplain second
}

fn() {
    bbplain third
}

do_foo_append() {
    bbplain fourth
}
```

Результатом запуска do_foo будет вывод строк

```
recipename do_foo: first
recipename do_foo: second
recipename do_foo: third
recipename do_foo: fourth
```

Переопределения и операторы с переопределением могут применяться не только в задачах, но и в любых shell-функциях. Для просмотра созданных функций после переопределений служит команда bitbake -e recipename.

3.5.2. Функции Python в стиле BitBake

Эти функции на языке Python выполняются BitBake или другими функциями Python с помощью bb.build.exec_func().

```
python some_python_function () {
    d.setVar("TEXT", "Hello World")
    print d.getVar("TEXT")
}
```

Поскольку модули Python bb и os уже импортированы, импортировать их не нужно. В функциях этого типа хранилище данных (d) является глобальной переменной и доступно всегда.

Выражения с переменными (например, ${X}) не преобразуются в функциях Python, чтобы позволить свободно устанавливать значения переменных в преобразуемые выражения без преждевременного преобразования. Если нужно преобразовать переменную внутри функции Python, следует применять d.getVar("X") или d.expand().

Как и в shell-функциях можно применять переопределения и операторы с переопределением.

```
python do_foo_prepend() {
    bb.plain("first")
}
python do_foo() {
    bb.plain("second")
}

python do_foo_append() {
    bb.plain("third")
}
```

Результатом запуска do_foo будет вывод строк

```
recipename do_foo: first
recipename do_foo: second
recipename do_foo: third
```

Для просмотра созданных функций после переопределений служит команда bitbake -e recipename.

3.5.3. Функции Python

Эти функции пишутся на языке Python и выполняются кодом Python. Примерами служат вспомогательные функции, предназначенные для вызова из встроенных или иных функций Python.

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

```
        return "dependency"
SOMECONDITION = "1"
DEPENDS = "${@get_depends(d)}"
```

Это будет приводить к включению в DEPENDS значения dependencywithcond.

Функции Python могут принимать параметры, хранилище данных BitBake автоматически не предоставляется и должно передаваться в параметре функции, модули bb и os доступны автоматически и не нужно импортировать их.

3.5.4. Функции в стиле Bitbake и обычные функции Python

Ниже перечислены основные различия функций Python в стиле BitBake и обычных функций, определяемых с def.

- Задачами могут быть только функции в стиле BitBake.
- Переопределения и операторы с переопределением доступны только в функциях со стилем BitBake.
- Только обычные функции Python могут принимать параметры и возвращать значения.
- Флаги переменных (такие как [dirs], [cleandirs], [lockfiles]) могут применяться только в функциях стиля BitBake.
- Функции в стиле BitBake создают отдельный сценарий ${T}/run.function-name.pid для своего запуска и журнальный файл ${T}/log.function-name.pid при выполнении как задачи. Обычные функции Python выполняются как встроенные (inline) и не создают файлов в ${T}.
- Обычные функции Python вызываются с использованием стандартного синтаксиса Python, а функции в стиле BitBake обычно являются задачами и вызываются напрямую из BitBake, но могут вызываться из кода Python с помощью функции bb.build.exec_func(). Например, bb.build.exec_func("my_bitbake_style_function", d).

  Функцию bb.build.exec_func() можно применять также для запуска shell-функций из Python. Если нужно запустить такую функцию в задаче до функции Python можно применить вспомогательную родительскую (parent) функцию Python, которая использует bb.build.exec_func() а затем выполнит код Python.

  Для обнаружения ошибок при вызове bb.build.exec_func() можно перехватывать bb.build.FuncFailed. Функциям в метаданных (задания и классы) не следует самим активировать bb.build.FuncFailed. Вместо этого можно просматривать bb.build.FuncFailed как индикатор ошибки при вызове функции путём активизации исключения. Например, активизация bb.fatal() будет захватывать bb.build.exec_func() и вызывать в ответ bb.build.FuncFailed.

Простота обычных функций Python делает их предпочтительней функций в стиле BitBake, если не нужны специфические возможности функций в стиле BitBake. Обычные функции Python появились в метаданных позднее функций в стиле BitBake и старый код более часто применяет bb.build.exec_func().

3.5.5. Анонимные функции Python

Иногда полезно устанавливать переменные и выполнять другие операции в процессе разбора. Для этого можно определять специальные функции Python, названные анонимными, которые выполняются в конце разбора. Например, приведённые ниже функция устанавливает переменную по условию.

```
python () {
    if d.getVar('SOMEVAR') == 'value':
        d.setVar('ANOTHERVAR', 'value2')
}
```

Другим способом является использованием для анонимных функций имени __anonymous.

Анонимные функции Python всегда применяются в конце анализа независимо от места их определения. При наличии множества анонимных функций в задании они выполняются в порядке определения.

```
python () {
    d.setVar('FOO', 'foo 2')
}

FOO = "foo 1"
python () {
    d.appendVar('BAR', ' bar 2')
}

BAR = "bar 1"
```

Приведённый пример концептуально эквивалентен приведённым ниже назначениям.

```
FOO = "foo 1"
BAR = "bar 1"
FOO = "foo 2"
BAR += "bar 2"
```

FOO будут иметь значение "foo 2", а BAR - "bar 1 bar 2". Как и во втором варианте, значения, установленные для переменных в анонимных функциях, доступны для задач, которые всегда запускаются после анализа.

Переопределения и операторы с переопределением, такие как _append, применяются до выполнения анонимных функций. В приведённом ниже примере FOO получит в результате значение "foo from anonymous".

```
FOO = "foo"
FOO_append = " from outside"

python () {
    d.setVar("FOO", "foo from anonymous")
}
```

Методы, которые можно применять с анонимными функциями, описаны в разделе 3.11. Функции, которые можно вызывать из Python. Другой метод выполнения кода Python в процессе разбора описан в параграфе 3.1.14. Преобразование переменных Python.

3.5.6. Гибкое наследование для функций класса

С помощью методов копирования и использования EXPORT_FUNCTIONS BitBake поддерживает экспорт функций из класса так, что функция класса выглядит принятой по умолчанию, но может вызываться даже если наследующему класс заданию нужно определить свою версию функции.

Для понимания преимуществ такого решения рассмотрим базовый сценарий, где класс определяет функцию задачи, а задание наследует этот класс, наследуя и функцию определённой в нем задачи. Если нужно в задании можно добавить начало и конец функции с помощью операторов _prepend и _append или полностью переопределить функцию. Однако при переопределении функции не будет возможности вызвать вариант этой функции из класса. EXPORT_FUNCTIONS обеспечивает механизм, позволяющей функции из задания вызвать исходный вариант функции из класса.

Для решения такой задачи нужно выполнить приведённые ниже условия.

- Класс должен определять функцию в форме classname_functionname. Например, для класса bar.bbclass и функции do_foo определение будет иметь вид bar_do_foo.
- Класс должен включать оператор EXPORT_FUNCTIONS в форме EXPORT_FUNCTIONS functionname. Для предыдущего примера это будет иметь вид EXPORT_FUNCTIONS do_foo.
- Нужно соответствующим образом вызвать функцию из задания. Для приведённого примера заданию следует вызвать bar_do_foo. Предположим, что do_foo является shell-функцией и переменная EXPORT_FUNCTIONS указана как было описано выше. Функция в задании может вызвать функцию класса по условию, задав

```
do_foo() {
        if [ somecondition ] ; then
                bar_do_foo
        else
                # Do something else
        fi
}
```

  Для вызова изменённой в задании функции применяется do_foo.

При выполнении этих условий задание может выбирать между исходным и измененным в задании вариантом функции. Если условия не выполнены, будут доступна лишь одна из функций.

3.6. Задачи

Задачи являются исполняемыми блоками BitBake, служащими этапами, которые BitBake может выполнять для задания. Задачи поддерживаются только в заданиях и классах (.bb и включённые или наследуемые ими файлы). По соглашению имена задач имеют префикс do_.

3.6.1. Представление функции задаче

Задачи являются функциями оболочки или функциями Python в стиле BitBake, которые представлены задаче с помощью команды addtask. Эта команда может также описывать зависимости от других задач. Ниже приведен пример определения задачи с указанием зависимостей.

```
python do_printdate () {
    import time
    print time.strftime('%Y%m%d', time.gmtime())
}
addtask printdate after do_fetch before do_build
```

Первым аргументом addtask является имя функции, представляемой задаче. Если имя не начинается с do_, этот префикс добавляется неявно в соответствии с принятым соглашением.

В примере задача do_printdate становится зависимостью do_build, которая выполняется по умолчанию (т. е. будет выполняться командой bitbake, если явно не указана иная задача). Кроме того, do_printdate становится зависимостью do_fetch. Запуск задачи do_build будет приводить к выполнению сначала do_printdate.

Если попытаться выполнить приведённый выше пример, можно будет увидеть, что задача do_printdate запускается только при первой сборке задания по команде bitbake. Это связано с тем, что после этого BitBake считает задачу «не устаревшей» (up-to-date). Если нужно повторять задачу каждый раз при экспериментальных сборках, можно «состарить» её с помощью флага переменной [nostamp] в форме do_printdate[nostamp] = "1". Можно также явно запускать задачу с опцией -f в форме bitbake recipe -c printdate -f. В этом случае префикс do_ не требуется.

Может возникнуть вопрос о практичности применения addtask без указания зависимости как addtask printdate. В этом случае (если зависимости не были заданы иначе) единственным способом запуска задачи будет bitbake recipe -c printdate. Можно использовать задачу do_listtasks для перечисления определённых в задании задач в форме bitbake recipe -c listtasks. Зависимости между задачами рассмотрены в разделе 3.10. Зависимости, а флаги переменных - в разделе 3.7. Флаги переменных.

3.6.2. Удаление задачи

Задачи можно не только добавлять, но и удалять командой deltask, например, deltask printdate. При удалении с помощью deltask задачи, имеющей зависимости, эти зависимости не включаются заново. Предположим, что имеются 3 задачи do_a, do_b, do_c и do_c зависит от do_b, а та - от do_a. Если в этом случае удалить do_b, неявная зависимость между do_c и do_a (через do_b) будет утеряна и зависимости do_c не обновятся для включения do_a. В результате do_c может быть запущена раньше do_a. Если нужно сохранить зависимости, следует применять флаг переменной [noexec] для отключения задачи без её удаления, в форме do_b[noexec] = "1".

3.6.3. Передача информации в среду сборки задачи

При запуске задачи BitBake строго контролирует среду сборки, чтобы нежелательное «загрязнение» сборочной машины не мешало процессу сборки. По умолчанию BitBake очищает среду для включения лишь тех переменных

которые экспортируются или указаны в «белом списке» для обеспечения согласованности и воспроизводимости сборки. Можно отключить такую «очистку» с помощью переменной BB_PRESERVE_ENV.

Для передачи чего-либо в выполняемую задачу нужно выполнить две операции, описанных ниже.

1. Указать BitBake загрузку нужных переменных из среды в хранилище данных с помощью переменных BB_ENV_WHITELIST и BB_ENV_EXTRAWHITE. Например, для запрета доступа системы сборки в каталог $HOME/.ccache можно включить в «белый список» переменную среды CCACHE_DIR для её добавления в хранилище данных BitBack.

   ```
   export BB_ENV_EXTRAWHITE="$BB_ENV_EXTRAWHITE CCACHE_DIR"
   ```

2. Указать BitBake экспорт включённого в хранилище данных в окружение каждой запущенной задачи. Загрузка из среды в хранилище (п. 1) лишь делает переменную доступной и для её экспорта в окружение запускаемых задач нужно использовать соответствующую команду в local.conf или файле конфигурации дистрибутива.

   ```
   export CCACHE_DIR
   ```

   Побочным эффектом этого является включение переменной в зависимости процесса сборки таких объектов как контрольные суммы setscene. Если это ведёт к ненужной переборке задач, можно внести переменную в «белый список», чтобы код setscene игнорировал зависимость при создании контрольных сумм.

Иногда полезна возможность получать информацию из исходной среды выполнения. Bitbake сохраняет её в специальной переменной BB_ORIGENV. Эта переменная возвращает объект хранилища данных, который можно запросить стандартными операторами хранилища, такими как getVar(, False). Этот объект полезен, например, для нахождения исходной переменной DISPLAY.

```
origenv = d.getVar("BB_ORIGENV", False)
bar = origenv.getVar("BAR", False)
```

Приведённый пример возвращает переменную BAR исходной среды.

3.7. Флаги переменных

Флаги переменных (varflag) помогают управлять функциями и зависимостями задач. BitBake читает и записывает varflag в хранилище, как показано ниже.

```
variable = d.getVarFlags("variable")
self.d.setVarFlags("FOO", {"func": True})
```

При работе с varflag применяется обычный синтаксис за исключением переопределений. Иными словами, можно добавлять флаги переменных в начале и в конце varflags (параграф 3.1.13. Синтаксис флагов переменных). В BitBake определён набор флагов для заданий и классов. Задачи поддерживают флаги, контролирующие функциональность.

- *[cleandirs]* указывает пустые каталоги, которые нужно создать до запуска задачи. Имеющиеся каталоги удаляются и вместо них создаются пустые.
- *[depends]* управляет зависимостями между задачами (см. DEPENDS и параграф 3.10.5. Зависимости между задачами).
- *[deptask]* управляет зависимостями при сборке (см. DEPENDS и параграф 3.10.2. Зависимости при сборке).
- *[dirs]* указывает каталоги, которые нужно создать до запуска задачи. Имеющиеся каталоги сохраняются. Указанный последним каталог становится текущим для задачи.
- *[lockfiles]* задаёт один или множество файлов блокировки для работы задачи. Файлом блокировки может владеть лишь одна задача и при попытке блокировать уже заблокированный файл задача будет остановлена до снятия блокировки. Можно применять этот флаг для согласования взаимного исключения задач.
- *[noexec]* значение 1 помечает задачу как пустую и не требующую выполнения. Флаг можно использовать для установки задач в качестве заменителей зависимостей и запрета выполнения ненужных задач в задании.
- *[nostamp]* значение 1 отключает создание BitBake файла штампа для задачи в результате чего задача будет исполняться всякий раз. Любые задачи, зависящие (возможно косвенно) от задачи [nostamp] также будут выполняться, что может удлинить сборку и поэтому требуется осторожная установка флага.
- *[number_threads]* ограничивает для задачи число одновременных потоков. Этот флаг полезен при работе с многоядерными процессами, когда некоторые задачи следует ограничивать по расходу ресурсов. Флаг number_threads работает подобно переменной BB_NUMBER_THREADS, но для одной задачи.

  Флаг устанавливается глобально. Например, для ограничения задачи do_fetch двумя потоками можно указать do_fetch[number_threads] = "2". Установка флага в задании (не глобально) может привести к непредсказуемому поведению. Установка значения больше BB_NUMBER_THREADS не будет оказывать влияния.
- *[postfuncs]* содержит список функций для вызова по завершении задачи.
- *[prefuncs]* содержит список функций для вызова перед выполнением задачи.
- *[rdepends]* управляет зависимостями между задачами во время работы (см. переменные RDEPENDS и RRECOMMENDS, а также параграф 3.10.5. Зависимости между задачами).
- *[rdeptask]* (см. переменные RDEPENDS и RRECOMMENDS, а также параграф (см. переменные RDEPENDS и RRECOMMENDS, а также параграф 3.10.3. Зависимости при работе).
- *[recideptask]* при установке вместе с recrdeptask указывает задачу для проверки дополнительных зависимостей.
- *[recrdeptask]* (см. переменные RDEPENDS и RRECOMMENDS, а также параграф (см. переменные RDEPENDS и RRECOMMENDS, а также параграф 3.10.4. Рекурсивные зависимости.

- *[stamp-extra-info]* дополнительные данные, добавляемые в конце штампа задачи. Например, OE использует флаг для машинозависимых задач.
- *[umask]* задаёт значение для работы задачи.

Некоторые флаги полезны для расчёта подписей переменных (см. 2.8. Контрольные суммы (подписи)).

- *[vardeps]* задаёт разделённый пробелами список дополнительных переменных, добавляемых в зависимости переменных для расчёта подписи. Добавление переменной в этот список полезно, например, при ссылке функции на переменную способом, который не позволяет BitBake определить эту ссылку автоматически.
- *[vardepsexclude]* задаёт разделённый пробелами список переменных, которые следует исключить из зависимостей переменных при расчёте подписи.
- [vardepvalue] при установке задаёт BitBake игнорировать действительное значение переменной и применять взамен при расчёте подписи указанное значение.
- *[vardepvalueexclude]* задаёт разделённый символами | список строк для исключения из значения переменной при расчёте подписи.

3.8. События

BitBake позволяет устанавливать обработчики событий в файлах заданий и классов. События инициируются в определённых точках во время работы, например, в начале обработки указанного задания (.bb), при запуске указанной задачи, отказе или успешном выполнении задачи и т. п. Цель заключается в упрощении таких действий, как передача по электронной почте сообщений об интересующих событиях. Ниже представлен пример обработчика, печатающего имя события и содержимое переменной FILE.

```
addhandler myclass_eventhandler
python myclass_eventhandler() {
    from bb.event import getName
    print("The name of the Event is %s" % getName(e))
    print("The file we run for is %s" % d.getVar('FILE'))
}
myclass_eventhandler[eventmask] = "bb.event.BuildStarted bb.event.BuildCompleted"
```

В этом примере маска событий (eventmask) установлена так, что обработчик видит лишь события BuildStarted и BuildCompleted. Обработчик вызывается при каждом событии, соответствующем маске. Определена глобальная переменная e, которая представляет текущее событие. Метод getName(e) возвращает имя вызвавшего обработчик события, глобальное хранилище доступно через переменную d. В устаревшем коде для доступа к хранилищу применяли e.data, однако сейчас от этого отказались в пользу d.

Контекст хранилища данных соответствует событию. Например, события BuildStarted и BuildCompleted происходят перед выполнением каких-либо задач и будут находиться в глобальном пространстве имён., где нет метаданных, специфичных для задания. Эти события происходят в основном процессе сборщика/сервера, а не в каком-либо рабочем контексте. Поэтому все изменения в хранилище будут видны другим событиям сборщика/сервера, но не будут доступны за пределами данной сборки или в каком-либо рабочем контексте. События задач происходят в конкретных задачах и поэтому имеют специфичное для задания и задачи содержимое. Эти события происходят в рабочем контексте и отбрасываются по завершении задачи.

Ниже описаны основные события, которые могут происходить при стандартной сборке.

- *bb.event.ConfigParsed()* происходит при анализе базовой конфигурации (bitbake.conf, base.bbclass и все глобальные операторы INHERIT). Можно видеть множество таких событий при разборе базовых конфигураций в каждой сборке или при изменении и повторном анализе конфигурации на сервере. Однако для любого конкретного хранилища данных происходит лишь одно такое событие. Если в хранилище установлен обработчик событий BB_INVALIDCONF, конфигурация анализируется повторно и возникает новое событие, позволяющее метаданным обновить конфигурацию.
- *bb.event.HeartbeatEvent()* происходит регулярно (1 раз в секунду, интервал можно изменить с помощью переменной BB_HEARTBEAT_EVENT). Атрибут time содержит значение time.time() в момент события. Событие полезно для таких действий, как мониторинг состояния системы.
- *bb.event.ParseStarted()* происходит, когда BitBake собирается начать анализ заданий. Атрибут total показывает число заданий, которые BitBake планирует анализировать.
- *bb.event.ParseProgress()* происходит в процессе анализа. Атрибут current указывает число анализируемых заданий, как и атрибут total.
- *bb.event.ParseCompleted()* происходит по завершении анализа. Атрибуты cached, parsed, skipped, virtuals, masked и errors содержат статистику анализа.
- *bb.event.BuildStarted()* происходит при запуске новой сборки. BitBake генерирует множество событий BuildStarted (1 на конфигурацию) при включённой поддержке нескольких конфигураций (multiconfig).
- *bb.build.TaskStarted()* происходит при запуске задачи. Атрибут taskfile указывает задание, которое вызвало задачу, атрибут taskname (имя задачи) включает префикс do_, logfile указывает место записи вывода задачи, а time - время запуска задачи.
- *bb.build.TaskInvalid()* происходит при попытке BitBake выполнить несуществующую задачу.
- *bb.build.TaskFailedSilent()* происходит при отказе задач setscene, когда подробный вывод не нужен.
- *bb.build.TaskFailed()* происходит при отказе обычных задач.
- *bb.build.TaskSucceeded()* происходит при успешном завершении задачи.
- *bb.event.BuildCompleted()* происходит при завершении сборки.

- *bb.cooker.CookerExit()* происходит при выключении сборщика/сервера BitBake и обычно воспринимается пользовательским интерфейсом как сигнал необходимости завершить работу.

Ниже приведен список событий, связанных с конкретными запросами к серверу. Обычно они связаны с передачей значительного объёма данных от сервера к другим частям BitBake, таким как интерфейс пользователя.

- *bb.event.TreeDataPreparationStarted();*
- *bb.event.TreeDataPreparationProgress();*
- *bb.event.TreeDataPreparationCompleted();*
- *bb.event.DepTreeGenerated();*
- *bb.event.CoreBaseFilesFound();*
- *bb.event.ConfigFilePathFound();*
- *bb.event.FilesMatchingFound();*
- *bb.event.ConfigFilesFound();*
- *bb.event.TargetsTreeGenerated().*

3.9. Варианты - механизм расширения класса

BitBake поддерживает две функции для создания из одного файла задания нескольких других заданий, которые являются собираемыми. Эти функции включаются переменными BBCLASSEXTEND и BBVERSIONS. Механизм этого расширения класса сильно зависит от реализации. Обычно переменные PROVIDES, PN и DEPENDS в задании классу расширения приходится менять. Примеры можно найти в классах OE-Core native, nativesdk и multilib.

- *BBCLASSEXTEND* содержит список разделенных пробелами классов, которые «расширяют» задание для каждого варианта. Например, BBCLASSEXTEND = "native" будет создавать второй вариант задания с наследованием класса native.
- *BBVERSIONS* позволяет одному заданию собирать несколько вариантов проекта на основе одного файла задания. Можно задать условные метаданные (с помощью OVERRIDES) для одной версии или заданного по условию диапазона версий. Например,

```
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"
```

Имя диапазона по умолчанию соответствует исходной версии задания. Например, в OE файл задания foo_1.0.0+.bb создает принятое по умолчанию имя 1.0.0+. Это полезно, поскольку имя диапазона не только помещается в переопределения, но и доступно метаданным для использования в переменной, определяющей версии базового задания для использования в пути поиска file:// (FILESPATH).

3.10. Зависимости

Для эффективной параллельной работы BitBake обрабатывает зависимости на уровне задач. Зависимости могут существовать между задачами одного или разных заданий. Ниже приведены примеры для обоих случаев.

- Внутри задания может потребоваться выполнить do_configure до запуска do_compile.
- Задача do_configure может требовать завершения do_populate_sysroot в другом задании для получения библиотек и заголовочных файлов.

В этом разделе описано несколько вариантов объявления зависимостей. Следует помнить, что при любом способе объявления зависимости существуют между задачами.

3.10.1. Зависимости внутри файла .bb

BitBake использует директиву addtask для управления внутренними зависимостями в файле задания. Например, addtask printdate after do_fetch before do_build задаёт зависимость задачи do_printdate от do_fetch task и зависимость do_build от do_printdate task. Для запуска задачи от неё. должна зависеть (напрямую или косвенно) другая задача, запланированная для запуска. Ниже приведено несколько примеров.

- Директива addtask mytask перед do_configure заставляет выполнить задачу do_mytask до запуска do_configure. Отметим, что do_mytask будет запускаться лишь при изменении контрольной суммы ввода с момента её предыдущего запуска. Изменение контрольной суммы ввода do_mytask вызывает также запуск do_configure.
- Директива addtask mytask после do_configure сама по себе не ведёт к запуску do_mytask, но задачу можно запустить вручную командой bitbake recipe -c mytask. Указание do_mytask в качестве зависимости для другой задачи, запуск которой запланирован, приведёт к запуску do_mytask, но в любом случае это произойдёт. после do_configure.

3.10.2. Зависимости при сборке

BitBake использует переменную DEPENDS для управления зависимостями при сборке. Флаг [deptask] для задач указывает для каждого элемента DEPENDS задачу, которая должна быть выполнена до запуска данной задачи. Например, do_configure[deptask] = "do_populate_sysroot" говорит о необходимости выполнить do_populate_sysroot для каждого элемента DEPENDS перед запуском для него задачи do_configure.

3.10.3. Зависимости при работе

BitBake использует переменные PACKAGES, RDEPENDS и RRECOMMENDS для управления зависимостями при работе. В переменной PACKAGES указаны пакеты, используемые при работе, и каждый из них может иметь зависимости RDEPENDS и RRECOMMENDS. Флаг [rdeptask] для задач указывает для каждой зависимости при работе задачу, которая должна быть выполнена до запуска данной задачи. Например, do_package_qa[rdeptask] = "do_packagedata" указывает, что задача do_packagedata должна быть выполнена для каждого элемента RDEPENDS до запуска do_package_qa.

3.10.4. Рекурсивные зависимости

BitBake использует флаг [recrdeptask] для управления рекурсивными зависимостями задач. BitBake просматривает зависимости текущего задания при сборке и работе, а также зависимости между задачами и затем добавляет зависимости для указанной задачи. После этого BitBake рекурсивно работает с зависимостями задач, продолжая итерации до нахождения и добавления всех зависимостей.

Флаг [recrdeptask] чаще всего применяется в заданиях верхнего уровня, которым приходиться дожидаться завершения той или иной задачи «глобально». Например, в класс image.bbclass включена зависимость do_rootfs[recrdeptask] += "do_packagedata", указывающая, что задачи do_packagedata в текущем задании и во всех доступных (через зависимости) заданиях из задания для образа должны быть выполнены до запуска задачи do_rootfs.

Может возникнуть желание просмотра BitBake не только зависимостей этих задач, но и зависимостей при сборке и работе для зависимых от них задач. В этом случае нужно указать имена задач в виде do_a[recrdeptask] = "do_a do_b".

3.10.5. Зависимости между задачами

BitBake использует флаг [depends] в более общей форме для управления зависимостями между задачами. Эта форма позволяет проверять такие зависимости для конкретных задач вместо проверки данных в DEPENDS. Например, do_patch[depends] = "quilt-native:do_populate_sysroot" задаёт выполнение задачи do_populate_sysroot для цели quilt-native до запуска do_patch. Флаг [rdepends] работает аналогично но относится к целям в пространстве имён. при работе, а не к пространству имён. зависимостей при сборке.

3.11. Функции, которые можно вызывать из Python

BitBake предоставляет много функций, которые можно вызывать из функций Python. Они кратко описаны ниже.

3.11.1. Функции для доступа к переменным хранилища данных

Часто нужен доступ к переменных хранилища данных BitBake с использованием функций Python. Хранилище Bitbake имеет интерфейс API для такого доступа. Список поддерживаемых операций приведен ниже.

Операция	*Описание*
`d.getVar("X", expand)`	Возвращает значение переменной X, преобразуя значение при expand=True. Если переменной X нет, возвращает None.
`d.setVar("X", "value")`	Устанавливает для переменной X значение value.
`d.appendVar("X", "value")`	Добавляет value в конце переменной X. Ведёт себя как d.setVar("X", "value") при отсутствии переменной X.
`d.prependVar("X", "value")`	Добавляет value в начале переменной X. Ведёт себя как d.setVar("X", "value") при отсутствии переменной X.
`d.delVar("X")`	Удаляет переменную X из хранилища при её наличии.
`d.renameVar("X", "Y")`	Переименовывает переменную X в Y, если она существует.
`d.getVarFlag("X", flag, expand)`	Возвращает значение переменной X, преобразуя значение при expand=True. Если переменной X или флага нет, возвращает None.
`d.setVarFlag("X", flag, "value")`	Устанавливает для флага переменной X значение value.
`d.appendVarFlag("X", flag, "value")`	Добавляет value в конец флагов переменной X. Ведёт себя как d.setVarFlag("X", flag, "value") при отсутствии флага.
`d.prependVarFlag("X", flag, "value")`	Добавляет value в начало флагов переменной X. Ведёт себя как d.setVarFlag("X", flag, "value") при отсутствии флага.
`d.delVarFlag("X", flag)`	Удаляет из хранилища флаг переменной X.
`d.setVarFlags("X", flagsdict)`	Устанавливает флаг, заданный в параметре flagsdict, не сбрасывая имеющийся флаг (как addVarFlags).
`d.getVarFlags("X")`	Возвращает flagsdict флагов переменной X или None, если X нет.
`d.delVarFlags("X")`	Удаляет все флаги переменной X при её наличии.
`d.expand(expression)`	Преобразует переменную, указанную строкой expression. Ссылки не отсутствующие переменные не меняются. Например, d.expand("foo ${X}") преобразует строку foo ${X}, если переменной X не существует.

3.11.2. Прочие функции

Другие функции, доступные для вызова из Python можно найти в коде модуля bb (bitbake/lib/bb). Например, bitbake/lib/bb/utils.py включает функции общего пользования bb.utils.contains() и bb.utils.mkdirhier() из docstrings.

3.12. Контрольные суммы задач и Setscene

BitBake использует контрольные суммы (подписи) с setscene для решения вопроса запуска задач. Здесь рассматривается этот процесс на примере метаданных OE.

Контрольные суммы хранятся в каталоге STAMP. Для проверки контрольных сумм служит команда bitbake-dumpsigs, возвращающая данные подписи в читаемом формате, которые позволяют проверить входные данные, используемые системой сборки OE при генерации подписей. Команда позволяет, например, проверить sigdata задачи do_compile для приложения C (например, bash). Команда также показывает, что переменная CC является частью хэшируемого ввода и изменение этой переменной будет делать штамп недействительным и приведёт к перезапуску задачи do_compile.

Ниже перечислены переменные, связанные с контрольными суммами.

- *BB_HASHCHECK_FUNCTION* указывает имя функции, вызываемой setscene для проверки хэш-значений.
- *BB_SETSCENE_DEPVALID* задаёт функцию, вызываемую BitBake для определения необходимости выполнения зависимости setscene.
- *BB_SETSCENE_VERIFY_FUNCTION2* задаёт функцию, вызываемую для проверки списка задач, выполнение которых запланировано до вызова основной задачи.
- *BB_STAMP_POLICY* задаёт режим сравнения временных меток в штампах.
- *BB_STAMP_WHITELIST* указывает файлы штампов, просматриваемые при политике whitelist.
- *BB_TASKHASH* содержит хэш выполняемой задачи, возвращаемый активированным генератором подписей.
- *STAMP* указывает базовый путь для создания штампов.
- *STAMPCLEAN* задаёт базовый путь для создания штампов с возможностью использования шаблонов для сопоставления при операциях очистки.

3.13. Поддержка шаблонов в переменных

Поддержка использования шаблонов в переменных зависит от контекста. Например, в некоторых именах переменных и файлов можно ограниченно применять шаблоны % и *, а другие поддерживают синтаксис Python glob, fnmatch или регулярных выражений. Описания переменных, поддерживающих шаблоны, указывают имеющиеся ограничения.

Глава 4. Поддержка загрузки файлов

Модуль сборки (fetch) BitBake является автономной частью библиотечного кода, которая отвечает за извлечение исходных кодов и файлов с удалённых сайтов. Извлечение исходных кодов является одной из важнейших частей сборки, поэтому модуль играет важную роль в BitBake. Современный модуль сборки называется fetch2 и обеспечивает вторую версию API. Исходная версия устарела и была удалена. Все упоминания fetch в документе относятся к fetch2.

4.1. Загрузка (выборка)

Извлечение исходного кода или файлов занимает в BitBake несколько этапов. Код сборщика выполняет 2 основных функции - получение файлов (возможно кэшированных) и их распаковку в указанное место (иногда указанным способом). Получение и распаковка файлов иногда сопровождаются внесением правок (patch). Код, отвечающий за первую часть процесса (выборка) имеет вид

```
src_uri = (d.getVar('SRC_URI') or "").split()
fetcher = bb.fetch2.Fetch(src_uri, d)
fetcher.download()
```

Код организует экземпляр класса fetch, использующий список разделенных пробелами URL из переменной SRC_URI для вызова методов загрузки файлов. За организацией экземпляра класса fetch обычно следует код

```
rootdir = l.getVar('WORKDIR')
fetcher.unpack(rootdir)
```

Этот код распаковывает загруженные файлы в каталоги, указанные WORKDIR.

Для удобства именование в примерах соответствует переменным OE. Для проверки кода в работе следует использовать файл класса OE base.bbclass. Переменные SRC_URI и WORKDIR не заданы жёстко в коде сборщика, поскольку методы сбора могут вызываться (и вызываются) с разными именами переменных. В OE, например, код общего состояния (sstate) использует модуль fetch для выборки файлов sstate.

При вызове метода download() BitBake пытается преобразовать URL, просматривая исходные файлы в указанном ниже порядке.

- *Pre-mirror Sites.* BitBake при поиске исходных файлов сначала просматривает PREMIRRORS.
- *Source URI.* Если файлов не найдено, BitBake использует исходную ссылку URL (например, из SRC_URI).
- *Mirror Sites.* При неудаче BitBake обращается к зеркалам, заданным переменной MIRRORS.

Для каждого переданного ему URL сборщик вызывает субмодуль, обрабатывающий данный тип URL. Это может вызывать некоторую путаницу при представлении URL для переменной SRC_URI. Например,

```
http://git.yoctoproject.org/git/poky;protocol=git
git://git.yoctoproject.org/git/poky;protocol=http
```

В первом случае URL передаётся сборщику wget, который не понимает протокол git, поэтому корректным будет второй вариант, поскольку сборщик Git понимает, как использовать транспорт HTTP.

Ниже приведено несколько примеров использования «зеркал».

```
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 \
```

```
        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"
```

Следует отметить, что BitBake поддерживает кросс-URL и возможно «зеркало» репозитория Git на сервере HTTP в виде архива (tarball). Именно это делает отображение git:// в прошлом примере. Поскольку доступ через сеть может быть медленным, Bitbake поддерживает кэш загруженных из сети файлов. Все нелокальные исходные файлы (т. е. загруженные из Internet) помещаются в специальный каталог, указанный переменной DL_DIR.

Целостность файлов очень важна для воспроизводимости сборки. Для нелокальных архивов код сборщика может проверять контрольную сумму SHA-256 или MD5, чтобы убедиться в корректности загрузки. Эти контрольные суммы можно задать с помощью переменной SRC_URI и подходящих флагов, как показано ниже.

```
SRC_URI[md5sum] = "value"
SRC_URI[sha256sum] = "value"
```

Можно также указать контрольные суммы как параметр SRC_URI.

```
SRC_URI = "http://example.com/foobar.tar.bz2;md5sum=4a8e0f237e961fd7785d19d07fdb994d"
```

При наличии множества URI можно задать контрольные суммы напрямую (см. выше) или путём именования URL.

```
SRC_URI = "http://example.com/foobar.tar.bz2;name=foo"
SRC_URI[foo.md5sum] = 4a8e0f237e961fd7785d19d07fdb994d
```

После загрузки и проверки контрольной суммы в DL_DIR помещается штамп .done, который BitBake использует при последующих сборках для предотвращения ненужной загрузки и новой проверки контрольной суммы. Предполагается, что данные в локальном хранилище не могут повреждаться. Если это не так, могут возникать серьёзные проблемы.

При установленной переменной BB_STRICT_CHECKSUM все загрузки с неверной контрольной суммой вызывают ошибки. Можно использовать переменную BB_NO_NETWORK, чтобы все попытки доступа в сеть вызывали критическую ошибку. Это может быть полезно для проверки полноты зеркал и других целей.

4.2. Распаковка

Распаковка обычно происходит сразу после загрузки и для всех URL (кроме Git) BitBake применяет один метод. В URL можно указать поведение этапа распаковки, как показано ниже.

- *unpack* управляет распаковкой компонент URL. Используемое по умолчанию значение 1 включает распаковку, а при значении 0 файлы остаются запакованными.
- *dos* применяется для файлов .zip и .jar, задавая символы перевода строки DOS в текстовых файлах.
- *basepath* указывает процессу распаковки вырезание указанных каталогов архива при распаковке.
- *subdir* задаёт распаковку указанного URL в указанный каталог внутри корневого каталога.

Распаковка может автоматически обрабатывать файлы .Z, .z, .gz, .xz, .zip, .jar, .ipk, .rpm. .srpm, .deb и .bz2, а также различные комбинации расширений архивов.

Как было отмечено, сборщик Git имеет свой метод распаковки, оптимизированный для работы с деревьями Git. Обычно этот метод клонирует дерево в каталог. Процесс завершается использованием ссылок, поэтому нужна лишь одна копия метаданных Git.

4.3. Сборщики

Префикс URL определяет используемый BitBake субмодуль сборщика. Эти субмодули описаны ниже.

4.3.1. Сборщик локальных файлов (file://)

Этот субмодуль обслуживает URL типа file://. Указанное в URL имя файла может быть относительным или абсолютным. Для относительных путей используется переменная FILESPATH как PATH при поиске файлов. Если файл не найден, предполагается, что он доступен в DL_DIR на момент вызова метода download(). Если указан каталог, он распаковывается целиком. Ниже приведены примеры URL с относительным и абсолютным путём.

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

4.3.2. Сборщик HTTP/FTP wget (http://, ftp://, https://)

Этот сборщик извлекает файлы с серверов web или FTP, используя утилиту wget. Исполняемый файл и параметры задаются переменной FETCHCMD_wget, для которой по умолчанию заданы разумные значения. Сборщик поддерживает параметр downloadfilename, позволяющий указать имя загруженного файла. Это полезно для предотвращения конфликтов имён. в DL_DIR. Ниже приведены примеры обрабатываемых сборщиком URL.

```
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.handhelds.org/home/you/secret.plan"
```

Разделение параметров URL точкой с запятой может приводить к неоднозначности разбора URL с такими символами.

```
SRC_URI = "http://abc123.org/git/?p=gcc/gcc.git;a=snapshot;h=a5dd47"
```

В таких URL следует заменять точку с запятой символом &, как показано ниже.

```
SRC_URI = "http://abc123.org/git/?p=gcc/gcc.git&a=snapshot&h=a5dd47"
```

Это работает в большинстве случаев, поскольку идентичную трактовку & в URL рекомендует консорциум W3C[1]. Отметим, что природа URL позволяет задать имя для загруженного файла, как показано ниже.

[1]World Wide Web Consortium.

```
SRC_URI = "http://abc123.org/git/?p=gcc/gcc.git&a=snapshot&h=a5dd47;downloadfilename=myfile.bz2"
```

4.3.3. Сборщик CVS (cvs://)

Этот субмодуль обслуживает выбор файлов из системы контроля версий CVS. Переменные настройки указаны ниже.

- *FETCHCMD_cvs* - имя исполняемого файла для команды cvs (обычно cvs).
- *SRCDATE* - дата выборки исходного кода CVS. Значение now служит для обновления при каждой сборке.
- *CVSDIR* - задаёт временное место хранения выборки (обычно DL_DIR/cvs).
- *CVS_PROXY_HOST* - имя, используемое в параметре proxy= для команды cvs.
- *CVS_PROXY_PORT* - номер порта, используемый в параметре proxyport= для команды cvs.

Наряду с обычным для URL указанием имени пользователя и пароля можно указать перечисленные ниже параметры.

- *"method"* задаёт протокол для связи с сервером CVS (по умолчанию pserver). При установке ext BitBake проверяет параметр rsh и устанавливает CVS_RSH. Можно применять dir для локальных каталогов.
- *"module"* задаёт выбираемый модуль (обязательно).
- *"tag"* описывает тег CVS TAG, который следует использовать для выбора (по умолчанию пусто).
- *"date"* задаёт дату. При отсутствии параметра используется SRCDATE из конфигурации. Значение now задаёт обновление при каждой сборке.
- *"localdir"* служит для переименования модуля (фактически переименовывается каталог для распаковки). Модуль помещается в каталог, указанный относительно CVSDIR.
- *"rsh"* используется вместе с параметром method.
- *"scmdata"* вызывает сохранение метаданных CVS в архиве, который сборщик создает при установке keep. Архив распаковывается в рабочий каталог. По умолчанию метаданные CVS удаляются.
- *"fullpath"* задаёт нахождение получаемой выборки на уровне модуля (по умолчанию) или ниже.
- *"norecurse"* заставляет сборщик выбирать указанный каталог без рекурсии подкаталогов.
- *"port"* указывает порт для соединения с сервером CVS.

Ниже приведены два примера.

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

4.3.4. Сборщик SVN (svn://)

Этот сборщик извлекает исходный код из системы контроля версий Subversion, используя исполняемый файл, указанный переменной FETCHCMD_svn (по умолчанию svn). Временный каталог задаёт SVNDIR (обычно DL_DIR/svn).

Параметры сборщика перечислены ниже.

- *"module"* задаёт имя модуля svn для выборки (обязательно).
- *"path_spec"* задаёт каталог для записи указанного модуля svn.
- *"protocol"* задаёт используемый протокол (по умолчанию svn). В случае svn+ssh добавляется параметр ssh.
- *"rev"* указывает выбираемый выпуск исходного кода.
- *"scmdata"* оставляет каталоги .svn во время компиляции при установке keep (по умолчанию удаляются).
- *"ssh"* - при установке protocol=svn+ssh может служить для указания программы ssh, применяемой svn.
- *"transportuser"* устанавливает имя пользователя для транспорта при необходимости (по умолчанию пусто). Имя пользователя для транспорта отличается от имени в URL, передаваемом команде subversion.

Ниже приведены примеры использования svn.

```
SRC_URI = "svn://myrepos/proj1;module=vip;protocol=http;rev=667"
SRC_URI = "svn://myrepos/proj1;module=opie;protocol=svn+ssh"
SRC_URI = "svn://myrepos/proj1;module=trunk;protocol=http;path_spec=${MY_DIR}/proj1"
```

4.3.5. Сборщик Git (git://)

Субмодуль извлекает код из системы управления Git. Сборщик создает клон удалённого репозитория в каталоге GITDIR (обычно DL_DIR/git2). Затем этот клон перемещается на этапе распаковки в рабочий каталог при выборе конкретной ветви. Это делается с использованием вариантов и ссылки для минимизации дублирования данных на диске и ускорения распаковки. Используется исполняемый файл, заданный переменной FETCHCMD_git.

Сборщик поддерживает перечисленные ниже параметры.

- *"protocol"* задаёт используемый для извлечения файлов протокол (по умолчанию git, если установлено имя хоста). Если имя хоста не задано, применяется протокол file. Можно использовать также http, https, ssh и rsync.
- *"nocheckout"* отменяет проверку исходного кода при распаковке, если задано значение 1. Это применяется при наличии другой программы проверки кода. По умолчанию 0.
- *"rebaseable"* указывает, что для восходящего репозитория Git поддерживается rebase. Следует устанавливать значение 1, если выпуски могут отделяться от ветвей. В таком случае архив зеркала источника выполняется по выпуску, что ведёт к потере эффективности. «Перебазирование» восходящего репозитория Git может

приводить к исчезновению из него текущего выпуска. Опция напоминает сборщику о необходимости сохранить локальный кэш для использования в будущем. По умолчанию параметр имеет значение 0.

- *"nobranch"* говорит сборщику, что не нужно проверять контрольные суммы SHA, если установлено значение 1 (по умолчанию 0). Следует устанавливать эту опцию в заданиях, указывающих фиксацию (commit), которая действительна для тега, а не для ветви.
- *"bareclone"* указывает сборщику, что нужно клонировать код в целевой каталог без выбора рабочего дерева с предоставлением лишь необработанных (raw) метаданных Git. Предполагается также параметр nocheckout.
- *"branch"* указывает клонируемые ветви дерева Git (по умолчанию master). Число параметров branch соответствует числу параметров name.
- *"rev"* указывает выбираемый выпуск (revision). По умолчанию master.
- *"tag"* указывает выбираемый тег. Для корректного преобразования тегов нужен доступ в сеть, поэтому теги зачастую не применяются. Для Git параметры tag и rev обеспечивают одинаковое поведение.
- *"subpath"* ограничивает выбор конкретным путём в дереве. По умолчанию дерево выбирается целиком.
- *"destsuffix"* указывает путь для размещения выборки (по умолчанию git/).
- *"usehead"* разрешает локальным URL git:// использовать текущую ветвь HEAD в качестве выпуска с AUTOREV. Параметр usehead подразумевает отсутствие ветвления и работает только с протоколом file://.

Примеры URL приведены ниже.

```
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"
```

4.3.6. Сборщик субмодулей Git (gitsm://)

Этот субмодуль наследует сборщик Git, расширяя его для выборки субмодулей. SRC_URI передаётся сборщику Git, как описано выше. При переключении между git:// и gitsm:// требуется очистка задания. Сборщик субмодулей Git не является полной реализацией и имеет ряд известных проблем с некорректным использованием инфраструктуры зеркал. Кроме того, извлекаемые коды субмодулей не видны инфраструктуре лицензирования и архивирования кода.

4.3.7. Сборщик ClearCase (ccrc://)

Этот субмодуль извлекает код из репозиториев ClearCase. Для использования сборщика нужно указать корректные значения SRC_URI, SRCREV и PV. Например,

```
SRC_URI = "ccrc://cc.example.org/ccrc;vob=/example_vob;module=/example_module"
SRCREV = "EXAMPLE_CLEARCASE_TAG"
PV = "${@d.getVar("SRCREV", False).replace("/", "+")}"
```

Сборщик использует в качестве клиента rcleartool или cleartool. Опции оператора SRC_URI приведены ниже.

- *vob* - имя ClearCase VOB, которое должно иметь в начале символ /. Параметр является обязательным.
- *module* - модуль в выбранном VOB с символом / в начале.

  Опции module и vob объединяются для создания правила загрузки (load). Например, комбинация vob и module из SRC_URI в приведённом выше примере даёт load /example_vob/example_module.
- *proto* указывает протокол http или https.

По умолчанию сборщик создает спецификацию конфигурации. Для её записи в отличное от принятого по умолчанию место следует использовать переменную CCASE_CUSTOM_CONFIG_SPEC в задании. При этом будет утеряна функциональность переменной SRCREV, однако она по-прежнему будет служить меткой архива после выборки.

Следует отметить некоторые аспекты поведения сборщика.

- При использовании cleartool для входа в систему не требуется дополнительных действий.
- При работе с rcleartool от имени конкретного пользователя нужна команда rcleartool login до запуска сборщика.

4.3.8. Сборщик Perforce (p4://)

Этот субмодуль извлекает код из системы управления Perforce, используя исполняемый файл, указанный переменной FETCHCMD_p4 (по умолчанию p4). Временный рабочий каталог задаёт переменная P4DIR (по умолчанию DL_DIR/p4).

Для использования сборщика нужно указать корректные значения SRC_URI, SRCREV и PV. Программа p4 способная использовать конфигурационный файл, определённый в переменной окружения P4CONFIG, для указания URL и порта сервера Perforce, имени пользователя и пароля, если они не указаны в задании. При отказе от использования P4CONFIG или явном указании переменных из P4CONFIG можно задать значение P4PORT с URL и номером порта сервера, а также указать имя пользователя и пароль в переменной задания SRC_URI.

Ниже приведен пример с использованием P4CONFIG и выборкой Head Revision.

```
SRC_URI = "p4://example-depot/main/source/..."
SRCREV = "${AUTOREV}"
PV = "p4-${SRCPV}"
S = "${WORKDIR}/p4"
```

В следующем примере задание указывает URL, порт, имя пользователя и пароля для выбора Revision по метке Label.

```
P4PORT = "tcp:p4server.example.net:1666"
SRC_URI = "p4://user:passwd@example-depot/main/source/..."
SRCREV = "release-1.0"
PV = "p4-${SRCPV}"
S = "${WORKDIR}/p4"
```

В задании всегда следует устанавливать S = "${WORKDIR}/p4".

4.3.9. Сборщик Repo (repo://)

Этот сборщик извлекает код из google-repo, сохраняя файлы в каталог REPODIR (по умолчанию, DL_DIR/repo).

Сборщик поддерживает несколько параметров, указанных ниже.

- *"protocol"* задаёт протокол для извлечения манифеста репозитория (по умолчанию git).
- *"branch"* указывает ветвь или тег для извлечения файлов (по умолчанию master).
- *"manifest"* указывает файл манифеста (по умолчанию default.xml).

Ниже приведены два примера URL для сборщика.

```
SRC_URI = "repo://REPOROOT;protocol=git;branch=some_branch;manifest=my_manifest.xml"
SRC_URI = "repo://REPOROOT;protocol=file;branch=some_branch;manifest=my_manifest.xml"
```

4.3.10. Другие сборщики

Имеются также субмодули Bazaar (bzr://), Mercurial (hg://), npm (npm://), OSC (osc://), Secure FTP (sftp://), Secure Shell (ssh://), Trees с использованием Git Annex (gitannex://), для которых ещё нет документации.

Глава 5. Глоссарий переменных

В этой главе приведены основные переменные, используемые BitBake с описанием их назначения и содержимого.

- Переменные, связанные только с BitBake, описание которых ограничивается этим контекстом.
- Используемые в BitBake переменные с такими же именами в других системах (например, YP и OE). В этом случае описывается расширенная функциональность.
- Отсутствующие в BitBake переменные других систем, которые использует BitBake.

A

ASSUME_PROVIDED

Список имён. заданий (значения PN), которые BitBake не пытается собирать, считая их уже собранными. В OpenEmbedded-Core переменная ASSUME_PROVIDED указывает естественные (native) инструменты, которые не нужно собирать. Примером служит git-native, при указании которого используется программа git на хосте сборки.

B

B

Каталог, в котором BitBake выполняет операции в процессе сборки задания.

BB_ALLOWED_NETWORKS

Список разделенных пробелами хостов, которые сборщику разрешено применять для извлечения исходного кода.

- Этот список используется лишь при отсутствии или нулевом значении переменной BB_NO_NETWORK.
- Ограниченно поддерживается шаблон * в именах. Например, вместо git.gnu.org, ftp.gnu.org и foo.git.gnu.org можно указать BB_ALLOWED_NETWORKS = "*.gnu.org". Символ * можно указывать лишь в начале имени перед точкой. Например, можно указать *.foo.bar, но не *aa.foo.bar.
- Зеркала, не указанные в списке хостов, пропускаются с записью события в журнал.
- Попытка доступа в сеть, не указанную в списке хостов, вызывает отказ.

Полезно использовать BB_ALLOWED_NETWORKS вместе с PREMIRRORS. Добавление нужно хоста в PREMIRRORS обеспечивает извлечение кода из разрешённого источника и позволяет избежать ошибок, связанных с отсутствием хоста в SRC_URI. Это обусловлено тем, что сборщик не пытается использовать хосты из SRC_URI после извлечения кода из PREMIRRORS.

BB_CONSOLELOG

Задаёт путь к журнальному файлу, куда пользовательский интерфейс BitBake записывает вывод сборки.

BB_CURRENTTASK

Имя текущей работающей задачи без префикса do_.

BB_DANGLINGAPPENDS_WARNONLY

Определяет обработку в BitBake ситуаций, когда файл добавления (.bbappend) не имеет соответствующего файла задания (.bb). Это часто возникает при рассинхронизации уровней (например, oe-core встречает версию задания, для которой старого файла уже нет, а другой уровень ещё не обновил версию задания). В такой ситуации отказ является самым безопасным решением.

BB_DEFAULT_TASK

Принятая по умолчанию задача для использования в случаях, когда ничего не указано (например, после опции -c). Имя задачи не следует указывать с префиксом do_.

BB_DISKMON_DIRS

Отслеживает доступное пространство и inode при сборке, позволяя контроль по этим параметрам (по умолчанию отключён). Переменная задаётся в форме BB_DISKMON_DIRS = "<action>,<dir>,<threshold> […]", где

<action>

ABORT - незамедлительное прерывание сборки при превышении порога.

STOPTASKS - остановка сборки после завершения текущих задач в случае превышения порога.

WARN - вывод предупреждения и продолжение сборки. Последующие предупреждения выдаются в соответствии с переменной BB_DISKMON_WARNINTERVAL, которая должна быть определена.

<dir>

Любые каталоги для мониторинга, разделённые пробелами. Если каталоги расположены на одном устройстве, отслеживается лишь первый.

<threshold>

Минимальное пространство на диске или/и минимальное число inode. Можно применять суффиксы G (Гбайт), M (Мбайт), K (Кбайт, принято по умолчанию). Не следует использовать GB, MB или KB.

```
BB_DISKMON_DIRS = "ABORT,${TMPDIR},1G,100K WARN,${SSTATE_DIR},1G,100K"
BB_DISKMON_DIRS = "STOPTASKS,${TMPDIR},1G"
BB_DISKMON_DIRS = "ABORT,${TMPDIR},,100K"
```

Первый пример работает только при установке переменной BB_DISKMON_WARNINTERVAL и заставляет систему сборки немедленно прерывать процесс, если свободное пространство в ${TMPDIR} становится меньше 1 Гбайта или число inode ниже 100 Кбайт. Поскольку указаны 2 каталога, система сборки будет выдавать предупреждение при снижении свободного места в ${SSTATE_DIR} меньше 1 Гбайт или inode меньше 100 Кбайт. Следующие предупреждения будут выдаваться с интервалами, заданными BB_DISKMON_WARNINTERVAL.

Второй пример будет останавливать сборку по завершении задач, когда свободное место в ${TMPDIR} станет меньше 1 Гбайт. Число inode не отслеживается.

Третий пример задаёт немедленное прерывание сборки при числе свободных inode в ${TMPDIR} меньше 100 Кбайт без отслеживания свободного места в каталоге.

BB_DISKMON_WARNINTERVAL

Задаёт интервал предупреждений о нехватке места на диске и свободных inode. Для использования переменной нужно задать BB_DISKMON_DIRS с действием WARN. В процессе сборки при снижении свободного места будут выдаваться предупреждения с заданным интервалом. Если при установке BB_DISKMON_DIRS = "WARN" интервал не указан, используется BB_DISKMON_WARNINTERVAL = "50M,5K". При задании переменной в файле конфигурации применяется форма BB_DISKMON_WARNINTERVAL = "<disk_space_interval>,<disk_inode_interval>".

<disk_space_interval>

Интервал свободного пространства в G (Гбайт), M (Мбайт) или K (Кбайт). Не используйте GB, MB или KB.

<disk_inode_interval>

Интервал свободных inode в G (Гбайт), M (Мбайт) или K (Кбайт). Не используйте GB, MB или KB.

```
BB_DISKMON_DIRS = "WARN,${SSTATE_DIR},1G,100K"
BB_DISKMON_WARNINTERVAL = "50M,5K"
```

Эти переменные заставляют BitBake выдавать предупреждения каждый раз, когда свободное место в каталоге ${SSTATE_DIR} снижается на 50 Мбайт или число свободных inode - на 5 Кбайт. Предупреждения будут выдаваться после того, как свободное место станет меньше 1 Гбайт или число свободных inode - меньше 100 Кбайт.

BB_ENV_WHITELIST

Задаёт внутренний список переменных, которые пропускаются из внешней среды в хранилище данных BitBake. Если переменная не задана (принято по умолчанию), пропускаются переменные BBPATH, BB_PRESERVE_ENV, BB_ENV_WHITELIST и BB_ENV_EXTRAWHITE. Для работы переменной её нужно установить во внешней среде.

BB_ENV_EXTRAWHITE

Задаёт список дополнительных переменных для пропуска из внешней среды в хранилище BitBake. Этот список дополняет BB_ENV_WHITELIST. Для работы переменной её нужно установить во внешней среде.

BB_FETCH_PREMIRRORONLY

Значение 1 заставляет модуль сборщика BitBake искать файлы только по переменной PREMIRRORS (без SRC_URI и MIRRORS).

BB_FILENAME

Имя файла задания, владеющего текущей выполняемой задачей. Например, при выполнении задачи do_fetch в задании my-recipe.bb переменная BB_FILENAME будет содержать /foo/path/my-recipe.bb.

BB_GENERATE_MIRROR_TARBALLS

Заставляет помещать архивы репозиториев Git (включая метаданные Git) в каталог DL_DIR. Из соображений производительности по умолчанию создание и сохранение архивов отключено в BitBake. Для включения нужно задать BB_GENERATE_MIRROR_TARBALLS = "1".

BB_HASHCONFIG_WHITELIST

Указывает переменные, исключаемые из расчёта контрольной суммы базовой конфигурации, используемой для решения вопроса об использовании кэша. Одним из способов решения вопроса о повторном использовании метаданных является контрольная сумма переменных в хранилище базовой конфигурации BitBake. Есть переменные, которые обычно не нужно учитывать в контрольной сумме. Например, TIME и DATE меняются постоянно, поэтому при их учёте BitBake просто не сможет повторно использовать кэш.

BB_HASHBASE_WHITELIST

Указывает переменные, исключаемые из данных контрольных сумм и зависимостей. Типичным примером такой переменной служит путь к сборке. Вывод BitBake не должен зависеть (и обычно не зависит) от каталога сборки.

BB_HASHCHECK_FUNCTION

Имя функции, вызываемой на этапе setscene при выполнении задачи для проверки списка хэш-значений задач. Функция возвращает список задач для выполнения. На этом этапе цель заключается в быстрой проверке работы данной функции setscene. Проще проверить функции по списку за один проход, нежели вызывать множество отдельных задач. Возвращаемый список может быть не совсем точным и данная задача setscene может потом завершиться отказом. Повышение точности возвращаемых данных обеспечивает рост эффективности сборки.

BB_INVALIDCONF

Используется с событием ConfigParsed для запуска повторного анализа базовых метаданных (т. е. всех заданий). Событие ConfigParsed может устанавливать переменную для повтора анализа. Нужно быть осторожным, чтобы не создать петлю.

BB_LOGFMT

Задаёт имя для журнальных файлов, сохраняемых в ${T}. По умолчанию переменная не задана и файлы именуются в форме log.{task}.{pid}.

BB_NICE_LEVEL

Позволяет BitBake работать с указанным приоритетом (уровень nice). Системные полномочия обычно позволяют BitBake понижать свой приоритет, но не повышать его снова. См. также описание BB_TASK_NICE_LEVEL.

BB_NO_NETWORK

Отключает доступ в сеть модулям выборки BitBake. В этом случае попытка доступа в сеть будет вызвать ошибку. Запрет полезен для тестирования зеркал исходного кода, сборки без доступа в Internet и при работе за некоторыми межсетевыми экранами.

BB_NUMBER_THREADS

Максимальное число задач, которые BitBake следует запускать параллельно. В системах с большим числом процессорных ядер значением этой переменной разумно устанавливать удвоенное число ядер.

BB_NUMBER_PARSE_THREADS

Задаёт число потоков, используемых BitBake при анализе. По умолчанию совпадает с числом процессорных ядер.

BB_ORIGENV

Содержит копию исходного внешнего окружения, в котором работает BitBake. Копия делается до применения фильтров хранилища BitBake. Эта переменная является обычным объектом хранилища данных.

BB_PRESERVE_ENV

Отключает «белый список» и пропускает все переменные из внешней среды в хранилище BitBake. Чтобы переменная работала, её нужно установить во внешней среде.

BB_RUNFMT

Указывает имя исполняемых сценариев (файлы run), сохраняемых в ${T}. По умолчанию переменная не задана и файлы именуются в форме run.{task}.{pid}.

BB_RUNTASK

Имя выполняемой в данный момент задачи с префиксом do_.

BB_SCHEDULER

Задаёт планировщик, используемый BitBake и может принимать одно из трёх значений:

- *basic* - базовая модель с числовым упорядочением задач по мере из разбора;
- *speed* - сначала выполняются задачи, от которых зависит больше других задач (принято по умолчанию);
- *completion* - планировщик пытается завершить данное задание после начала его сборки.

BB_SCHEDULERS

Указывает настраиваемый планировщик для импорта. Эти планировщики должны выводиться из класса RunQueueScheduler. Выбор планировщика определяется переменной BB_SCHEDULER.

BB_SETSCENE_DEPVALID

Задаёт функцию, вызываемую BitBake для определения необходимости выполнения зависимости setscene. При запуске задачи setscene нужно знать, какие из её зависимостей также следует запускать. Выполнение зависимостей существенно зависит от метаданных. Заданная переменной функция возвращает True или False.

BB_SETSCENE_VERIFY_FUNCTION2

Задаёт функцию, вызываемую для проверки списка выполнения запланированных задач перед запуском основной задачи. Функция вызывается при наличии у BitBake списка задач setscene, которые были запущены и завершились отказом или успехом. Функция позволяет проверить осмысленность списка задач. Если в BitBake был запланирован пропуск задачи, результат функции может заставить BitBake запустить эту задачу в зависимости от метаданных, определяемых обстоятельствами.

BB_SIGNATURE_EXCLUDE_FLAGS

Список флагов переменных (varflag), которые можно безопасно исключить из контрольных сумм и данных зависимостей для ключей в хранилище. При генерации контрольных сумм и данных зависимостей для ключей установленные флаги ключа обычно учитываются (см. раздел 3.7. Флаги переменных).

BB_SIGNATURE_HANDLER

Задаёт имя обработчика подписей, используемого BitBake, который определяет способ создания и обработки файлов штампов, встраивания в них подписей и создания этих подписей. Можно добавить обработчик путём внедрения класса, выведенного из SignatureGenerator, в глобальное пространство имён.

BB_SRCREV_POLICY

Определяет поведение сборщика при взаимодействии с системами управления версиями и динамическими выпусками кода. Переменная полезна при работе без сети. Поддерживается два варианта:

- *cache* - сохраняется полученное ранее значение вместо запроса системы управления версиями;
- *clear* - система управления версиями запрашивается каждый раз (без кэширования, принято по умолчанию).

BB_STAMP_POLICY

Задаёт режим сравнения временных меток в файлах штампов:

- *perfile* - сравниваются только метки в одном задании (принято по умолчанию);
- *full* - сравниваются метки для всех зависимостей;
- *whitelist* - похоже на full, но метки сравниваются для заданий из переменной BB_STAMP_WHITELIST.

Правила для штампов по большей части устарели с введением задач setscene.

BB_STAMP_WHITELIST

Список файлов, для которых временные метки штампов сравниваются в режиме whitelist (BB_STAMP_POLICY).

BB_STRICT_CHECKSUM

Задаёт более строгую проверку контрольных сумм для нелокальных URL. BitBake будет возвращать ошибку при обнаружении нелокального URL, для которого не задана хотя бы одна контрольная сумма.

BB_TASK_IONICE_LEVEL

Позволяет настроить приоритет ввода-вывода (I/O) для задачи. При тестировании Autobuilder могут возникать случайные ошибки связанные с насыщением I/O во время тайм-аутов QEMU, которые можно исключить настройкой приоритетов. Переменная работает подобно BB_TASK_NICE_LEVEL, но относится только к операциям I/O.

```
BB_TASK_IONICE_LEVEL = "class.prio"
```

Для class по умолчанию используется значение 2 (best effort), но можно задать 1 (realtime) или 3 (idle). Для режима realtime требуются полномочия superuser. Для prio можно использовать значения от 0 (высший приоритет) до 7 (низший), по умолчанию установлено 4. Для установки prio не требуется особых полномочий.

Для работы настроек приоритета I/O нужна поддержка планировщика CFQ[1] для блочного устройства. Для выбора планировщика служит приведённая ниже команда, где вместо device нужно указать реальное устройство (sda, sdb).

```
$ sudo sh -c "echo cfq > /sys/block/device/queu/scheduler
```

BB_TASK_NICE_LEVEL

Позволяет конкретной задаче изменить свой приоритет (уровень nice). Эту переменную можно применять с переопределениями задач для установки приоритета конкретных задач. Например, в Yocto Project autobuilder эмуляция QEMU в образах получает более высокий приоритет, нежели задачи сборки, чтобы не возникало тайм-аутов на загруженных системах.

BB_TASKHASH

В исполняемой задаче эта переменная содержит хэш задачи, возвращаемый активным генератором подписей.

BB_VERBOSE_LOGS

Задаёт повышенную детализацию вывода BitBake при сборке, включая команды echo и вывод shell-сценариев.

[1]Completely Fair Queuing - беспристрастное управление очередями.

BB_WORKERCONTEXT

Указывает выполнение задачи в текущем контексте (1). Значение не устанавливается, если задача находится в контексте сервера в процессе синтаксического анализа или обработки события.

BBCLASSEXTEND

Позволяет расширить задание для сборки вариантов программы. Такими вариантами для заданий из метаданных OpenEmbedded-Core являются native, такие как quilt-native (копия Quilt для работы в системе сборки), cross, такие как gcc-cross (компилятор для машину сборки, создающий двоичные файлы для целевой MACHINE), nativesdk, предназначенные для SDK вместо MACHINE, и mulitlibs в форме multilib:multilib_name. Сборка разных вариантов задания с минимальным объёмом кода обычно так же проста, как добавление переменной в задание. Ниже приведены два примера вариантов из метаданных OpenEmbedded-Core.

```
BBCLASSEXTEND =+ "native nativesdk"
BBCLASSEXTEND =+ "multilib:multilib_name"
```

Механизм BBCLASSEXTEND создает варианты задания, переписывая значения переменных и применяя переопределения, такие как _class-native. Например, для генерации native-версии задания в переменной DEPENDS заменяется foo на foo-native. Даже при использовании BBCLASSEXTEND задание собирается в один приём. Однократный разбор вносит некоторые ограничения, например, невозможно включить разные файлы в зависимости от варианта, поскольку операторы include обрабатываются при анализе задания.

BBDEBUG

Устанавливает уровень отладочного вывода BitBake, который можно увеличить опцией -D. Для работы переменной её нужно задать во внешней среде.

BBFILE_COLLECTIONS

Список имён. настроенных уровней для поиска других переменных BBFILE_*. Обычно каждый уровень добавляет своё имя в конце этой переменной в файле conf/layer.conf.

BBFILE_PATTERN

Переменная, преобразуемая для сопоставления с файлами из BBFILES в отдельном уровне. Переменная задаётся в файле conf/layer.conf и должна иметь имя уровня как суффикс (например, BBFILE_PATTERN_emenlow).

BBFILE_PRIORITY

Задаёт приоритет файлов задания в каждом уровне. Это полезно в ситуациях, где одно задание присутствует на нескольких уровнях. Переменная позволяет приоритизировать уровни с одинаковыми заданиями. Заданный переменной приоритет сохраняется независимо от версии задания (PV). Например, уровень может включать файл с большим значением PV, но меньшим приоритетом в BBFILE_PRIORITY и не будет выбран в результате. Большее значение переменной обеспечивает более высокий приоритет. Если переменная не задана, она устанавливается на основе зависимостей уровня (переменная LAYERDEPENDS). Если приоритет не задан для уровня без зависимостей, значение определяется минимальным имеющимся приоритетом +1 (или 1, если приоритетов не задано). Для просмотра настроенных уровней с их приоритетами служит команда bitbake-layers show-layers.

BBFILES

Разделённый пробелами список файлов заданий, используемых BitBake для сборки. При указании файлов можно использовать синтаксис Python glob.

BBINCLUDED

Разделённый пробелами список файлов, включённых анализатором BitBake при разборе текущего файла.

BBINCLUDELOGS

Включает вывод журнала задачи при сообщении об отказе задачи.

BBINCLUDELOGS_LINES

При установленной переменной BBINCLUDELOGS задаёт максимальное число выводимых строк из журнала задачи при сообщении об отказе. По умолчанию журнал задачи выводится целиком.

BBLAYERS

Список уровней, включённых в сборку. Задаётся в файле bblayers.conf внутри каталога сборки. Например,

```
BBLAYERS = " \
  /home/scottrif/poky/meta \
  /home/scottrif/poky/meta-yocto \
  /home/scottrif/poky/meta-yocto-bsp \
  /home/scottrif/poky/meta-mykernel \
  "
```

Здесь заданы 4 уровня, один из которых (meta-mykernel) является пользовательским.

BBLAYERS_FETCH_DIR

Указывает базовое местоположение уровней. Переменная используется с командой bitbake-layers layerindex-fetch.

BBMASK

Управляет исключением файлов заданий и дополнений из обработки BitBake, «пряча» ненужные файлы. BitBake игнорирует задания и дополнения, соответствующие выражениям маски. Значение переменной передаётся компилятору регулярных выражений Python, в маске применяется синтаксис Python Regular Expression (re). Сравнение выполняется для полных путей к файлам. Приведённый ниже пример обеспечивает игнорирование BitBake всех заданий и дополнений в каталоге meta-ti/recipes-misc/.

```
BBMASK = "meta-ti/recipes-misc/"
```

Если нужно скрыть множество каталогов или заданий, можно указать несколько фрагментов регулярных выражений, как показано ниже.

```
BBMASK += "/meta-ti/recipes-misc/ meta-ti/recipes-ti/packagegroup/"
BBMASK += "/meta-oe/recipes-support/"
BBMASK += "/meta-foo/.*/openldap"
BBMASK += "opencv.*\.bbappend"
BBMASK += "lzma"
```

Для исключения каталога следует указывать символ / в конце имени.

BBMULTICONFIG

Позволяет BitBake выполнить сборку со множеством конфигураций и указывает каждую конфигурацию (multiconfig). С помощью этой переменной можно задать BitBake сборку нескольких целей, каждая из которых имеет свою конфигурацию. Переменная задаётся в файле conf/local.conf, например, BBMULTIFONFIG = "configA configB configC". Каждый конфигурационный файл должен размещаться в каталоге сборки внутри каталога conf/multiconfig (например, build_directory/conf/multiconfig/configA.conf). Использование переменной в среде с поддержкой сборки нескольких конфигураций описано в параграфе 1.5.2.5. Сборка с несколькими конфигурациями.

BBPATH

Используется BitBake для поиска файлов классов (.bbclass) и конфигурации (.conf) подобно переменной окружения PATH. При запуске BitBake извне сборочного каталога нужно убедиться, что BBPATH указывает каталог сборки. Переменная устанавливается во внешнем окружении до запуска BitBake, как показано ниже.

```
$ BBPATH="build_directory"
$ export BBPATH
$ bitbake target
```

BBSERVER

Указывает сервер, на котором работает BitBake в режиме memory-resident, и работает только в этом случае.

BBTARGETS

Позволяет использовать файл конфигурации для добавления в команду заданий, которые нужно собрать.

BBVERSIONS

Позволяет собрать несколько версий программы по одному файлу задания. Можно также указать дополнительные метаданные с использованием механизма переопределения (OVERRIDES) для одной версии или именованного набора версий (см. раздел 3.9. Варианты - механизм расширения класса).

BITBAKE_UI

Служит для указания модуля UI, используемого BitBake (как опция -u в команде). Для работы переменной её нужно установить во внешней среде.

BUILDNAME

Имя сборки. По умолчанию имя задаёт временная метка начала сборки, но его можно изменить в метаданных.

BZRDIR

Каталог, в котором сохраняются выбранные файлы Bazaar.

C

CACHE

Каталог, используемый BitBake для кэширования метаданных, которые не нужно разбирать при каждом запуске.

CVSDIR

Каталог, в котором сохраняются выбранные файлы CVS.

D

DEFAULT_PREFERENCE

Задаёт незначительное смещение базы приоритета для выбора задания. Обычно для этой переменной устанавливается значение "-1" в заданиях для находящихся в разработке версий. Такое использование переменной ведёт к сборке по умолчанию стабильной версии задания при отсутствии PREFERRED_VERSION для сборки разрабатываемой версии.

Смещение в DEFAULT_PREFERENCE невелико и переопределяется значением BBFILE_PRIORITY, если эта переменная различается между двумя уровнями с разными версиями одного задания.

DEPENDS

Список зависимостей при сборке задания (другие задания).

DESCRIPTION

Длинное описание задания.

DL_DIR

Центральный каталог загрузки, используемый процессом сборки для хранения загруженных файлов. По умолчанию DL_DIR принимает любые файлы для «зеркалирования», кроме репозиториев Git. Если нужно архивировать файлы репозиториев Git, следует использовать переменную BB_GENERATE_MIRROR_TARBALLS.

E

EXCLUDE_FROM_WORLD

Предписывает BitBake исключить задание из сборки world (bitbake world), когда BitBake находит, анализирует и собирает все задания, найденные в каждом уровне, указанном в файле bblayers.conf. Для исключения задания следует указать значение 1 для этой переменной в задании. Указанные в переменной задания могут сохраняться в сборке world для выполнения зависимостей других заданий, т. е. добавление задания в переменную исключает лишь его явное включение в список заданий сборки world.

F

FAKEROOT

Содержит команду, используемую при работе сценариев оболочки в среде fakeroot. Переменная устарела и заменена переменными FAKEROOT*, описанными ниже.

FAKEROOTBASEENV

Список переменных среды, которые нужно установить при выполнении команды, заданной в FAKEROOTCMD, которая запускает процесс bitbake-worker в среде fakeroot.

FAKEROOTCMD

Команда, запускающая процесс bitbake-worker в среде fakeroot.

FAKEROOTDIRS

Список каталогов, создаваемых перед запуском задачи в среде fakeroot.

FAKEROOTENV

Список переменных окружения, устанавливаемых при запуске задачи в среде fakeroot (см. FAKEROOTBASEENV).

FAKEROOTNOENV

Список переменных окружения, устанавливаемых при запуске задачи вне среды fakeroot (см. FAKEROOTENV).

FETCHCMD

Определяет команду, выполняемую сборщиком BitBake при операции извлечения кода. При использовании переменной требуется задавать суффикс переопределения (например, FETCHCMD_git или FETCHCMD_svn).

FILE

Указывает текущий файл. BitBake устанавливает переменную в процессе анализа, а также при выполнении задания для идентификации файла.

FILESPATH
Указывает каталоги, используемые BitBake при поиске правок (patch) и файлов. Сборщик local использует эти каталоги при обработке URL file://. Переменная похожа на переменную среды PATH и содержит разделённые двоеточиями каталоги, которые просматриваются слева направо.

G

GITDIR
Каталог для хранения локальной копии репозитория Git при его клонировании.

H

HGDIR
Каталог для хранения файлов, выбранных из Mercurial.

HOMEPAGE
Web-сайт с информацией о программе, собираемой заданием.

I

INHERIT
Задаёт классы с глобальным наследованием. Анонимные функции классов не выполняются для базовой конфигурации и отдельных заданий, система сборки OE игнорирует изменения INHERIT в отдельных заданиях (см. параграф 3.4.5. Конфигурационная директива INHERIT).

L

LAYERDEPENDS
Список разделенных пробелами уровней, от которых зависит задание. Можно также указать конкретную версию уровня для зависимости, добавив её к имени после двоеточия (например, anotherlayer:3 будет сравниваться с LAYERVERSION_anotherlayer). BitBake возвращает ошибку при невыполнении зависимости или несовпадении версии. Переменная задаётся в файле conf/layer.conf с использованием имени задания в качестве суффикса (например, LAYERDEPENDS_mylayer).

LAYERDIR
При указании в layer.conf задаёт путь к текущему уровню. Переменная недоступна вне layer.conf и ссылки на неё. преобразуются сразу же при анализе файла.

LAYERDIR_RE
При указании в layer.conf задаёт путь к текущему уровню с экранированием (escape) для использования в регулярном выражении (BBFILE_PATTERN). Переменная недоступна вне layer.conf и ссылки на неё. преобразуются сразу же при анализе файла.

LAYERVERSION
Может задавать версию уровня одним числом. Переменную можно использовать в LAYERDEPENDS для другого уровня при указании зависимости от конкретной версии. Переменная задаётся в conf/layer.conf и должна содержать имя конкретного уровня в качестве суффикса (например, LAYERDEPENDS_mylayer).

LICENSE
Список лицензий для задания.

M

MIRRORS
Задаёт дополнительные пути для поиска исходных кодов. Система сборки сначала проверяет каталог загрузок, затем места, указанные в PREMIRRORS, восходящий репозиторий и зеркала, указанные MIRRORS.

MULTI_PROVIDER_WHITELIST
Позволяет отключить предупреждения BitBake при сборке двух отдельных заданий с одинаковым выводом. Обычно Bitbake выдаёт такие предупреждения, но в некоторых случаях подобная сборка имеет смысл (в частности, для пространства имён. virtual/*). Переменная содержит список провайдеров (например, имена заданий, virtual/kernel и т. п.).

O

OVERRIDES
BitBake использует OVERRIDES для управления переопределениями переменных после разбора заданий и файлов конфигурации. Ниже приведен простой пример переопределения на основе архитектуры машины.

```
OVERRIDES = "arm:x86:mips:powerpc"
```

Дополнительная информация приведена в разделе 3.3. Синтаксис условий.

P

P4DIR
Каталог с локальной копией хранилища Perforce при его выборке.

PACKAGES
Список пакетов, создаваемых заданием.

PACKAGES_DYNAMIC
Обещание соответствия пакета зависимостям при работе для дополнительных модулей из других заданий. Переменная на деле не выполняет этих зависимостей, а только говорит, что они должны быть выполнены. Например, если жёсткая зависимость при работе (RDEPENDS) от другого пакета выполняется при сборке через переменную PACKAGES_DYNAMIC, а пакет с модулем на деле не создаётся, пакет не сможет работать.

PE
Эпоха для задания (по умолчанию не задана). Переменная применяется для обновления при изменении схемы версий несовместимым со старыми версиями способом.

PERSISTENT_DIR
Каталог, используемый BitBake для хранения данных между сборками. В частности, это данные, используемые BitBake API, а также сервером и службой PR.

PF
Имя задания или пакета с номером версии и выпуска (например, eglibc-2.13-r20+svnr15508/ или bash-4.2-r1/).

PN

Имя задания.

PR

Выпуск (revision) задания.

PREFERRED_PROVIDER

Определяет предпочтительное задание, если несколько заданий предоставляют один и тот же элемент. Следует всегда применять эту переменную с суффиксом из имени предоставляемого задания (значение PN из задания), для которого указывается предпочтение. Например,

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_PROVIDER_virtual/xserver = "xserver-xf86"
PREFERRED_PROVIDER_virtual/libgl ?= "mesa"
```

PREFERRED_PROVIDERS

Определяет предпочтительное задание, если несколько заданий предоставляют один и тот же элемент. Функционально идентична переменной PREFERRED_PROVIDER, однако позволяет задать предпочтения для нескольких ситуаций в форме PREFERRED_PROVIDERS = "xxx:yyy aaa:bbb …". Это служит удобной заменой для

```
PREFERRED_PROVIDER_xxx = "yyy"
PREFERRED_PROVIDER_aaa = "bbb"
```

PREFERRED_VERSION

При наличии нескольких версий задания эта переменная определяет предпочтительное. Следует всегда применять эту переменную с суффиксом из имени задания (значение PN из задания) и устанавливать в задании нужное значение PV. Переменная PREFERRED_VERSION поддерживает ограниченное использование шаблона %, которому могут соответствовать любые символы. Например,

```
PREFERRED_VERSION_python = "2.7.3"
PREFERRED_VERSION_linux-yocto = "4.12%"
```

Символ % можно указывать только последним в строке.

PREMIRRORS

Указывает дополнительные пути получения исходного кода для BitBake. При поиске кода система сборки сначала проверяет локальные источники в каталоге загрузки. Если там ничего не найдено, проверяются места, указанные в переменной PREMIRRORS, восходящий источник, а затем MIRRORS в указанном порядке.

Обычно в переменной указываются предпочтительный сервер, например

```
PREMIRRORS_prepend = "\
git://.*/.* http://www.yoctoproject.org/sources/ \n \
ftp://.*/.* http://www.yoctoproject.org/sources/ \n \
http://.*/.* http://www.yoctoproject.org/sources/ \n \
https://.*/.* http://www.yoctoproject.org/sources/ \n"
```

В результате система сборки будет перехватывать запросы Git, FTP, HTTP, HTTPS и направлять на зеркало http://. Можно указать URL file:// для загрузки с локальных или общих сетевых дисков.

PROVIDES

Список псевдонимов, под которыми может быть известно определённое задание. По умолчанию значение PN для задания неявно включено в этот список. Если задание содержит PROVIDES, дополнительные псевдонимы являются синонимами задания и могут применяться для соблюдения зависимостей других заданий во время сборки, указанных в DEPENDS. Например, в файле libav_0.8.11.bb указано PROVIDES += "libpostproc" и в результате задание libav будет известно также под именем libpostproc.

Помимо представления заданий под другими именами, механизм PROVIDES также служит для реализации виртуальных целей, т. е. имён., соответствующих той или иной функциональности (например, ядро Linux). Задания, обеспечивающие такую функциональность, указывают виртуальную цель в PROVIDES, а зависящие от этой функциональности задания могут включать виртуальную цель в DEPENDS, оставляя выбор провайдера открытым.

Обычно виртуальные цели имеют имена вида virtual/function (например, virtual/kernel), где символ / является частью имени и не имеет синтаксического значения.

PRSERV_HOST

Хост и порт сетевой службы PR, например PRSERV_HOST = "localhost:0". Эту переменную нужно устанавливать при автоматическом запуске сервиса PR. В переменной PRSERV_HOST можно указать удалённую службу PR.

PV

Версия задания.

R

RDEPENDS

Список зависимостей пакета при работе (другие пакеты), которые должны быть установлены. Если пакет не найден при сборке, возникает ошибка. Поскольку переменная RDEPENDS применяется к собираемым пакетам, её всегда следует связывать с именем пакета. Предположим, например, сборку пакета разработки, зависящего от perl. В этом случае следует указать RDEPENDS_${PN}-dev += "perl". BitBake поддерживает указание версии в зависимостях. Хотя синтаксис зависит от формата пакетов, эта зависимость скрывается BitBake и в переменных RDEPENDS применяется синтаксис RDEPENDS_${PN} = "package (operator version)", где оператором может быть =, <, >, <=, >=. Например, для указания зависимости от пакета foo версии не ниже 1.2 можно использовать RDEPENDS_${PN} = "foo (>= 1.2)". Зависимости при сборке рассмотрены в описании переменной DEPENDS.

REPODIR

Каталог, в котором хранится локальная копия google-repo при синхронизации.

RPROVIDES

Список псевдонимов, которые имеет пакет. Псевдонимы полезны для выполнения зависимостей при работе пакетов как во время сборки, так и на целевой платформе (RDEPENDS). Как во всех переменных управления пакетами должно применяться переопределение по имени пакета, например, RPROVIDES_${PN} = "widget-abi-2"

RRECOMMENDS

Список пакетов, расширяющих применимость собираемого пакета, который не зависит напрямую от этих пакетов при сборке, но они нужны для расширения возможностей. Для задания зависимостей при работе служит переменная RDEPENDS. BitBake поддерживает указание версии в рекомендациях. Хотя синтаксис зависит от формата пакетов, эта зависимость скрывается BitBake и в переменных RRECOMMENDS применяется синтаксис RRECOMMENDS_${PN} = "package (operator version)" , где оператором может быть =, <, >, <=, >=. Например, для рекомендации пакета версии не ниже 1.2 можно использовать RRECOMMENDS_${PN} = "foo (>= 1.2)".

S

SECTION

Категория, к которой следует отнести пакет.

SRC_URI

Список исходных файлов (локальных или удалённых). Переменная указывает BitBake исходные файлы для сборки и способ их получения. Например, если заданию или файлу дополнения нужно извлечь архив из Internet, используется запись SRC_URI, указывающая этот архив. Если же нужно извлечь архив и добавить пользовательские файлы, переменная SRC_URI должна указывать все источники. Ниже перечислены поддерживаемые протоколы.

- *file://* - выборка файлов, обычно сопровождаемых метаданными, из локальных каталогов. Пути указываются относительно переменной FILESPATH.
- *bzr://* - выборка из репозитория Bazaar.
- *git://* - выборка из репозитория Git.
- *osc://* - выборка из репозитория OSC (OpenSUSE Build service).
- *repo://* - выборка из репозитория repo (Git).
- *http://* - выборка из Internet по протоколу HTTP.
- *https://* - выборка из Internet по протоколу HTTPS.
- *ftp://* - выборка из Internet по протоколу FTP.
- *cvs://* - выборка из репозитория CVS.
- *hg://* - выборка из репозитория Mercurial (hg).
- *p4://* - выборка из репозитория Perforce (p4).
- *ssh://* - выборка по протоколу ssh.
- *svn://* - *выборка из репозитория Subversion (svn)*.

Следует также отметить ряд опций:

- *unpack* - управляет распаковкой архивов (включена по умолчанию);
- *subdir* - указывает каталог для размещения файла ии распаковки архива (полезно для архивов без структуры каталогов);
- *name* - указывает имя, используемое для привязки к контрольной сумме SRC_URI, когда в SRC_URI указно несколько файлов.
- *downloadfilename* - имя для сохранения загруженного файла.

SRCDATE

Дата исходного кода, использованного для сборки, применяемая лишь для источников, полученных из SCM.

SRCREV

Выпуск (revision) исходного кода, использованного для сборки, применяемый лишь для источников, полученных из Subversion, Git, Mercurial и Bazaar. Если нужно собрать конкретный выпуск и избежать запроса к удалённому репозиторию при каждом разборе задания, следует указать в переменной полный идентификатор, а не просто тег.

SRCREV_FORMAT

Помогает создать корректное значение SRCREV при использовании множества управляемых источниками URL в переменной SRC_URI. Каждой компоненте SRC_URI даётся имя, указываемое в SRCREV_FORMAT. Например, URL можно назвать machine и meta, а переменная SRCREV_FORMAT может иметь вид machine_meta и в эти имена будут подставляться версии SCM. При необходимости можно добавлять заполнитель AUTOINC в начале возвращаемой строки.

STAMP

Задаёт базовый путь для файлов штампов задания. Путь к реальному файлу штампа создаётся преобразованием этой строки с добавлением в конце дополнительной информации.

STAMPCLEAN

Задаёт базовый путь для файлов штампов задания. В отличие от STAMP, переменная STAMPCLEAN может включать шаблоны для сопоставления с диапазоном файлов, которые должна удалить операция очистки, используемая BitBake для удаления старых штампов.

SUMMARY

Краткое описание задания (до 72 символов).

SVNDIR

Каталог для хранения выбранных файлов Subversion.

T

T

Указывает каталог, где BitBake хранит временные файлы (в основном сценарии и журналы вывода) при сборке конкретного задания.

TOPDIR

Указывает каталог сборки (BitBake автоматически устанавливает переменную).

Приложение A. Пример Hello World

A.1. BitBake Hello World

В этом простом примере показано, как создать новый проект и подходящие метаданные в контексте BitBake.

A.2. Получение BitBake

Информация о получении и установке приведена в разделе 1.4. Получение BitBake. Каталог BitBake имеет вид

```
$ ls -al
total 100
drwxrwxr-x. 9 wmat wmat  4096 Jan 31 13:44 .
drwxrwxr-x. 3 wmat wmat  4096 Feb  4 10:45 ..
-rw-rw-r--. 1 wmat wmat   365 Nov 26 04:55 AUTHORS
drwxrwxr-x. 2 wmat wmat  4096 Nov 26 04:55 bin
drwxrwxr-x. 4 wmat wmat  4096 Jan 31 13:44 build
-rw-rw-r--. 1 wmat wmat 16501 Nov 26 04:55 ChangeLog
```

```
drwxrwxr-x. 2 wmat wmat  4096 Nov 26 04:55 classes
drwxrwxr-x. 2 wmat wmat  4096 Nov 26 04:55 conf
drwxrwxr-x. 3 wmat wmat  4096 Nov 26 04:55 contrib
-rw-rw-r--. 1 wmat wmat 17987 Nov 26 04:55 COPYING
drwxrwxr-x. 3 wmat wmat  4096 Nov 26 04:55 doc
-rw-rw-r--. 1 wmat wmat    69 Nov 26 04:55 .gitignore
-rw-rw-r--. 1 wmat wmat   849 Nov 26 04:55 HEADER
drwxrwxr-x. 5 wmat wmat  4096 Jan 31 13:44 lib
-rw-rw-r--. 1 wmat wmat   195 Nov 26 04:55 MANIFEST.in
-rw-rw-r--. 1 wmat wmat  2887 Nov 26 04:55 TODO
```

A.3. Организация среды BitBake

Сначала нужно проверить возможность запуска BitBake. В каталоге BitBake введите команду

```
$ ./bin/bitbake --version
BitBake Build Tool Core version 1.23.0, bitbake version 1.23.0
```

На консоль будет выведена информация о версии, как показано выше.

Рекомендуется запускать BitBake из каталога программы, но можно также включить каталог BitBake в переменную PATH, предварительно посмотрев текущее значение с помощью команды echo $PATH. Затем следует добавить в начало этой переменной каталог BitBake. Например, для программы в каталоге /home/scott-lenovo/bitbake/bin следует использовать команду export PATH=/home/scott-lenovo/bitbake/bin:$PATH. После этого можно использовать команду bitbake из любого каталога.

A.4. Пример Hello World

Целью примера Hello World является иллюстрация применения концепций задач и уровней. Поскольку именно так работают OE и YP, использующие BitBake, это будет хорошей отправной точкой для начала работы. Дополнительную информацию можно получить из почтовой конференции http://lists.openembedded.org/mailman/listinfo/bitbake-devel.

Пример Hello World описывается ниже.

1. *Создание каталога для проекта* под названием hello в домашнем каталоге.

   ```
   $ mkdir ~/hello
   $ cd ~/hello
   ```

   Этот BitBake будет использовать для работы и в нем же можно хранить метаданные, нужные для BitBake.

2. *Запуск Bitbake.*

   ```
   $ bitbake
   The BBPATH variable is not set and bitbake did not
   find a conf/bblayers.conf file in the expected location.
   Maybe you accidentally invoked bitbake from the wrong directory?
   DEBUG: Removed the following variables from the environment:
   GNOME_DESKTOP_SESSION_ID, XDG_CURRENT_DESKTOP,
   GNOME_KEYRING_CONTROL, DISPLAY, SSH_AGENT_PID, LANG, no_proxy,
   XDG_SESSION_PATH, XAUTHORITY, SESSION_MANAGER, SHLVL,
   MANDATORY_PATH, COMPIZ_CONFIG_PROFILE, WINDOWID, EDITOR,
   GPG_AGENT_INFO, SSH_AUTH_SOCK, GDMSESSION, GNOME_KEYRING_PID,
   XDG_SEAT_PATH, XDG_CONFIG_DIRS, LESSOPEN, DBUS_SESSION_BUS_ADDRESS,
   _, XDG_SESSION_COOKIE, DESKTOP_SESSION, LESSCLOSE, DEFAULTS_PATH,
   UBUNTU_MENUPROXY, OLDPWD, XDG_DATA_DIRS, COLORTERM, LS_COLORS
   ```

   Основная часть вывода относится к переменным окружения, не связанным напрямую с BitBake. Однако первая фраза о переменной BBPATH и файле conf/bblayers.conf важна для понимания. При запуске BitBake выполняется поиск метаданных и переменная BBPATH говорит, где нужно их искать. Без этой переменной Bitbake не может найти конфигурационных файлов (.conf) и заданий (.bb), а также файла bitbake.conf.

3. *Настройка BBPATH.* В этом примере переменную BBPATH можно задать как выше было показано для PATH, но более эффективным способом является её установка в конфигурационном файле для каждого проекта.

   ```
   $ BBPATH="projectdirectory"
   $ export BBPATH
   ```

   В первой команде следует указать реальный каталог проекта и BitBake будет искать в нем метаданные. При указании проекта недопустимо использовать символ ~, поскольку BitBake не преобразует его.

4. *Запуск Bitbake.*

   ```
   $ bitbake
   ERROR: Traceback (most recent call last):
     File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 163, in wrapped
       return func(fn, *args)
     File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 173, in parse_config_file
       return bb.parse.handle(fn, data, include)
     File "/home/scott-lenovo/bitbake/lib/bb/parse/__init__.py", line 99, in handle
       return h['handle'](fn, data, include)
     File "/home/scott-lenovo/bitbake/lib/bb/parse/parse_py/ConfHandler.py", line 120, in handle
       abs_fn = resolve_file(fn, data)
     File "/home/scott-lenovo/bitbake/lib/bb/parse/__init__.py", line 117, in resolve_file
       raise IOError("file %s not found in %s" % (fn, bbpath))
   IOError: file conf/bitbake.conf not found in /home/scott-lenovo/hello

   ERROR: Unable to parse conf/bitbake.conf: file conf/bitbake.conf not found in /home/scott-lenovo/hello
   ```

   Из вывода видно, что BitBake не может найти в каталоге проекта файл conf/bitbake.conf, без которого не может работать.

5. *Создание файла conf/bitbake.conf.* Этот файл содержит многочисленные переменные, используемые BitBake для файлов метаданных и заданий. В нашем примере нужно создать файл в каталоге проекта и задать в нем

некоторые важные переменные BitBake. Пример файла доступен по ссылке http://git.openembedded.org/bitbake/tree/conf/bitbake.conf.

```
$ mkdir conf
```

Далее следует с помощью подходящего редактора создать в новом каталоге файл bitbake.conf.

```
PN  = "${@bb.parse.BBHandler.vars_from_file(d.getVar('FILE', False),d)[0] or 'defaultpkgname'}"

TMPDIR  = "${TOPDIR}/tmp"
CACHE   = "${TMPDIR}/cache"
STAMP   = "${TMPDIR}/${PN}/stamps"
T       = "${TMPDIR}/${PN}/work"
B       = "${TMPDIR}/${PN}"
```

Без переменной PN, переменные STAMP, T и B не позволят работать более чем одному заданию. Можно установить в PN значение, похожее на используемое OE и BitBake в принятом по умолчанию файле bitbake.conf, как показано выше, или обновлять вручную каждое задание, устанавливая PN. Может также потребоваться включение PN в определения переменных STAMP, T и B в файле local.conf.

Переменная TMPDIR указывает каталог, который BitBake использует для вывода сборки и промежуточных файлов, не являющихся кэшируемыми данными для процесса Setscene. Каталог временных файлов можно без опаски удалять перед повторной сборкой и сборочный процесс BitBake создаст его заново.

6. *Запуск Bitbake.* После создания файла conf/bitbake.conf можно снова ввести команду bitbake.

```
$ bitbake
ERROR: Traceback (most recent call last):
  File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 163, in wrapped
    return func(fn, *args)
  File "/home/scott-lenovo/bitbake/lib/bb/cookerdata.py", line 177, in _inherit
    bb.parse.BBHandler.inherit(bbclass, "configuration INHERITs", 0, data)
  File "/home/scott-lenovo/bitbake/lib/bb/parse/parse_py/BBHandler.py", line 92, in inherit
    include(fn, file, lineno, d, "inherit")
  File "/home/scott-lenovo/bitbake/lib/bb/parse/parse_py/ConfHandler.py", line 100, in include
    raise ParseError("Could not %(error_out)s file %(fn)s" % vars(), oldfn, lineno)
    ParseError: ParseError in configuration INHERITs: Could not inherit file classes/base.bbclass

ERROR: Unable to parse base: ParseError in configuration INHERITs: Could not inherit file classes/base.bbclass
```

Сейчас BitBake не может найти файл classes/base.bbclass и нужно создать его.

7. *Создание classes/base.bbclass.* BitBake использует файлы классов для хранения общего кода и функций. Для работы нужно иметь, как минимум, файл classes/base.bbclass. Класс base неявно наследуется каждым заданием. BitBake ищет класс в каталоге classes внутри проекта (hello/classes в этом примере).

```
$ cd $HOME/hello
$ mkdir classes
```

В новом каталоге нужно создать класс base.bbclass с единственной строкой addtask build. Минимальной задачей при запуске BitBake является выполнение do_build и в нашем примере ничего другого не требуется.

8. *Запуск Bitbake.* После создания файла classes/base.bbclass можно снова ввести команду bitbake.

```
$ bitbake

Nothing to do.  Use 'bitbake world' to build everything, or run 'bitbake --help' for usage information.
```

BitBake больше не выдаёт сообщений об ошибках, однако говорит о том, что делать при сборке нечего. Нужно подготовить задание, которое BitBake будет выполнять.

9. *Создание уровня.* Хотя в нашем элементарном примере можно обойтись без своего уровня, использование уровня в каждом проекте является хорошим тоном и позволяет отделить код от метаданных общего пользования, применяемых BitBake. Пазовем наш уровень mylayer. Уровень должен включать, по меньшей мере, файл задания и конфигурации уровня.

```
$ cd $HOME
$ mkdir mylayer
$ cd mylayer
$ mkdir conf
```

В каталоге conf создаём файл layer.conf, как показано ниже.

```
BBPATH .= ":${LAYERDIR}"

BBFILES += "${LAYERDIR}/*.bb"

BBFILE_COLLECTIONS += "mylayer"
BBFILE_PATTERN_mylayer := "^${LAYERDIR_RE}/"
```

Затем создаём файл задания printhello.bb в каталоге проекта, как показано ниже.

```
DESCRIPTION = "Prints Hello World"
PN = 'printhello'
PV = '1'

python do_build() {
   bb.plain("********************");
   bb.plain("*                  *");
   bb.plain("*  Hello, World!   *");
   bb.plain("*                  *");
   bb.plain("********************");
}
```

Файл содержит описание, имя и версию задания, а также задачу do_build, которая выводит сообщение "Hello World" на консоль.

10. *Запуск Bitbake с указанием цели.* Цель сборки BitBake и можно ввести команду для сборки.

```
$ cd $HOME/hello
$ bitbake printhello
ERROR: no recipe files to build, check your BBPATH and BBFILES?

Summary: There was 1 ERROR message shown, returning a non-zero exit code.
```

Сообщение об ошибке говорит, что BitBake не может найти задание, поскольку нет файла conf/bblayers.conf со списком уровней проекта.

11. *Создание файла conf/bblayers.conf*, указывающего уровни проекта. Файл должен размещаться в каталоге conf внутри проекта (hello/conf в нашем примере). Содержимое файла для нашего примера приведено ниже.

```
BBLAYERS ?= " \
  /home/<you>/mylayer \
  "
```

12. *Запуск Bitbake с указанием цели.* Файл конфигурации bblayers.conf создан и можно повторить команду.

```
$ bitbake printhello
Parsing recipes: 100% |##################################################################################|
Time: 00:00:00
Parsing of 1 .bb files complete (0 cached, 1 parsed). 1 targets, 0 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
NOTE: Preparing RunQueue
NOTE: Executing RunQueue Tasks
********************
*                  *
*  Hello, World!   *
*                  *
********************
NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
```

При повторном вводе команды bitbake printhello на консоли не появится приведённого выше текста, поскольку программа BitBake уже выполнила задачу do_build для printhello.bb и не будет её повторять. Если удалить каталог tmp с помощью команды bitbake -c clean printhello и повторить сборку, текст будет выведен снова.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru