Yocto Project Development Tasks Manual

Scott Rifenbark

Scotty's Documentation Services, INC
<srifenbark@gmail.com>

Этот документ основан на переводе *Yocto Project Development Tasks Manual* для выпуска 2.7.1. Свежие версии оригинальных документов можно найти на странице Yocto Project documentation. Размещённые там материалы более актуальны, нежели включённые в архивы пакета Yocto Project.

Оглавление

Глава 1. Руководство по разработке проектов в Yocto Project

1.1. Введение

В этом руководстве описаны процедуры разработки в среде Yocto Project (YP) образов Linux для встраиваемых систем и пользовательских приложений для работы на таких системах. Документ разбит на несколько разделов, описывающих связанные между собой процедуры.

Документ включает приведённые ниже описания.

- Процедуры, помогающие начать работу с YP, например, процедуры организации хоста для сборки и работы с репозиториями YP.
- Процедуры фиксации изменений в YP (улучшения, новые функции, исправления ошибок).
- Процедуры общего назначения при разработке образов и приложений с использованием YP, например, процедуры создания уровней, настройки образов, создания новых заданий (recipe) и т. п.

Глава 2. Настройка YP

В этой главе описаны процедуры подготовки к работе с YP, включая создание среды командной работы с использованием YP, настройку хоста для сборки, поиск репозиториев YP создание локального репозитория Git.

2.1. Создание среды для командной работы

Использование YP для командной работы на первый взгляд может показаться не очевидным. Одной из сильных сторон YP является гибкость системы. Вы можете настроить её для множества вариантов использования. Однако эта гибкость может вызывать сложности при настройке системы для работы большой команды.

В этом параграфе описана организация среды коллективной разработки и представлена процедура для решения этой задачи. Эта процедура реализуется на верхнем уровне и проверена на практике. Следует учитывать, что описанная процедура обеспечивает лишь стартовую точку и вам в любом случае придётся приложить усилия по настройке системы в соответствии со своими реальными потребностями и задачами.

1. *Определение состава команды разработчиков.* Определение участников разработки с использованием YP и их ролей. Это важно для выполнения пп. 2 и 3 при выборе оборудования и топологии.

   Ниже перечислены возможные роли участников процесса.

   - *Разработчик приложений* создаёт программы, работающие на базе имеющегося программного стека.
   - *Разработчик ядра системы* создаёт образы операционной системы.
   - *Инженер сборки* управляет автоматическими сборщиками (Autobuilder) и выпусками. Эта роль присутствует не всегда.
   - *Инженер тестирования* создаёт автоматизированные тесты и управляет ими для проверки качества приложений и операционной системы, а также соответствия стандартам.

2. *Подбор оборудования.* Соберите оборудование с учётом размера и состава команды. В идеальном варианте всем членам команды лучше работать в среде на основе поддерживаемого дистрибутива Linux. Рабочие станции должны быть достаточно производительными (например, два 6-ядерных процессора Xeon с 24 Гбайт ОЗУ и достаточным пространством на диске). Для тестирования производительности и использования Autobuilder потребуются высокопроизводительные компьютеры.

3. *Анализ аппаратной топологии оборудования.* После выбора оборудования для работы команды нужно разобраться с топологией среды разработки.

4. *Использование Git в качестве менеджера исходных кодов (SCM[1]).* Рекомендуется хранить метаданные (задания, файлы конфигурации, классы и т. п.) и все разрабатываемые программы под контролем системы SCM, совместимой с системой сборки OpenEmbedded (OE). Из числа SCM, поддерживаемых BitBake, команда YP настоятельно рекомендует выбрать Git. Это распределенная система с простым резервированием, позволяющая работать удалённо, а затем возвращаться в инфраструктуру.

   Дополнительная информация о BitBake представлена в [6].

   Сравнительно просто организовать службы Git и создать инфраструктуру похожую на http://git.yoctoproject.org, которая основана на серверной программе gitolite с использованием cgit для генерации web-интерфейса, позволяющего просматривать репозитории. Программа gitolite идентифицирует пользователей с помощью ключей SSH и обеспечивает управление доступом по ветвям для репозиториев.

   Организация этих служб выходит за рамки документа, но ниже приведены ссылки на документы, описывающие эти процессы.

   - Документация Git с описанием установки gitolite на сервере.
   - Описание Gitolite.
   - Интерфейсы и инструменты для Git.

5. *Организация машин для разработки приложений.* Как отмечено выше, разработчики приложений создают программы для работы на имеющемся программном стеке. Ниже приведены рекомендации по организации машин для такой работы.

   - Используйте собранный набор инструментов с нужным стеком программ и создавайте приложения для работы в этом стеке. Этот метод хорош для небольшого числа сравнительно изолированных приложений.
   - Поддерживайте инструменты кросс-разработки в актуальном состоянии. Это можно делать путём загрузки новых версий инструментальных пакетов или обновления с помощью механизма opkg. Выбор варианта зависит от ваших потребностей и принятой политики.
   - Используйте множество инструментальных пакетов, установленных локально в разных местах для работы с разными версиями.

6. *Организация машин для разработки ядра системы.* Как отмечено выше, разработчики ядра системы работают непосредственно с образами операционных систем. Ниже приведены рекомендации по организации машин для такой работы.

   - Настройте систему сборки OE [3] на рабочих станциях разработчиков, чтобы они могли создавать свои сборки и напрямую пересобирать образы.
   - Сохраняйте систему сборки неизменной и вносите свои изменения в своих уровнях на основе этой системы. Это обеспечит переносимость работы при обновлении системы или пакетов BSP[2].
   - Созданные уровни следует местно использовать всем разработчикам в соответствии с политикой настройки, определённой для проекта.

7. *Установка системы Autobuilder.* Средства автоматической сборки зачастую являются ядром среды разработки. Именно здесь собираются вместе и тестируются результаты отдельных разработчиков. На базе этой автоматизированной среды сборки и тестирования принимается решение о выпуске. Autobuilder также позволяет тестировать программные компоненты в стиле «непрерывной интеграции» с обнаружением и отслеживанием регрессии.

   На странице YP Autobuilder представлена подробная информация и ссылка на buildbot, который команда YP сочла подходящим для автоматического тестирования. Общедоступным примером является YP Autobuilder, используемый командой YP для тестирования проекта в целом. Свойства системы приведены ниже.

   - Указание фиксаций (commit), нарушающих сборку.

[1]Source Control Manager.
[2]Board Support Package - пакет поддержки плат.

- Заполнение кэша sstate [1], из которого разработчики могут извлекать данные без локальной сборки.
- Поддержка триггеров фиксаций, включающих сборки при новом представлении (commit).
- Возможность включать автоматизированную загрузку и тестирование образов в QEMU[1].
- Поддержка пошагового тестирования сборки и сборок «с нуля».
- Общий доступ к выводу сборки, позволяющий разработчикам выполнять тестирование и искать проблемы.
- Создание вывода, который можно использовать в выпусках системы.
- Возможность сборки по расписанию для эффективного использования ресурсов.

8. *Установка машин для тестов.* Используйте несколько высокопроизводительных систем для тестирования. Разработчики могут применять эти машины для масштабного тестирования, продолжая работу на своих системах.
9. *Правила документирования и поток изменений.* YP использует иерархическую структуру и модель извлечения. Имеются сценарии для создания и отправки pull-запросов (create-pull-request и send-pull-request). Эта модель соответствует другим проектам с открытым кодом, где сопровождающие отвечают за определённые области проекта, а один обрабатывает слияния на вершине дерева (top-of-tree).

   Можно также использовать коллективную модель выталкивания (push), поскольку gitolite одинаково легко поддерживает модели push и pull.

   Как в любой среде разработки важно документировать используемые правила, а также основные принципы проекта, чтобы они были понятны каждому. Хорошо также структурировать сообщения фиксации (commit), что обычно делается в описании принципов проекта. Чёткие сообщения фиксации нужны для понимания внесённых в проект изменений.

   При необходимости внесения изменений в ядро проекта следует как можно скорее поделиться ими с сообществом. Скорее всего другие члены сообщества также нуждаются в этих изменениях.
10. *Дополнительные рекомендации.*
    - Используйте Git в качестве системы управления исходными файлами.
    - Поддерживайте метаданные на уровнях, которые имеют смысл для вашей ситуации (см. раздел The Yocto Project Layer Model [1] и параграф 3.1. Уровни и их создание).
    - Разделяет метаданные и код проекта, используя разные репозитории Git (см. раздел Yocto Project Source Repositories [1] и параграф 2.3. Доступ к исходным файлам YP.
    - Создайте каталог для кэширования общих данных состояния (SSTATE_DIR). Например создайте кэш sstate на системе разработчиков и используйте одинаковые каталоги исходных кодов на их машинах.
    - Установите систему Autobuilder и поместите в неё кэш sstate и каталоги с исходными кодами.
    - Сообщество YP призывает отправлять исправления (patch) в проекте для устранения ошибок и расширения возможностей. Если вы представляете изменения, следуйте правилам фиксации (см. параграф 3.31.2. Представление изменений в YP.
    - Отправляйте изменения в ядре системы как можно скорей, поскольку с этими же проблемами могут столкнуться другие люди. Рекомендации и списки рассылок приведены в параграфе 3.31.2. Представление изменений в YP и разделе Mailing Lists [3].

2.2. Подготовка сборочного хоста

В этом параграфе описана настройка системы, которая будет служить сборочным хостом для разработки с использованием YP. Хост может быть Linux-машиной (рекомендуется) или машиной (Linux, Mac, Windows) с CROPS и контейнерами Docker. Хост с WSL[2] не подойдёт в качестве сборочного по причине несовместимости YP с WSL.

После установки YP на сборочном хосте дальнейшие шаги зависят от ваших задач. Ниже представлены ссылки на информацию для случаев работы с BSP и ядром Linux.

- *Разработка BSP* - раздел Preparing Your Build Host to Work With BSP Layers [5].
- *Разработка ядра* - раздел Preparing the Build Host to Work on the Kernel [7].

2.2.1. Подготовка сборочного хоста Linux

1. *Используйте поддерживаемый дистрибутив Linux.* Вам следует выбрать хост с текущей версией того или иного дистрибутива Linux. Рекомендуется применять свежий выпуск Fedora, openSUSE, Debian, Ubuntu или CentOS, поскольку эти дистрибутивы тестируются с YP и поддерживаются официально[3]. Список всех поддерживаемых дистрибутивов и детали работы с ними представлены в разделе Supported Linux Distributions [3] и на странице Distribution Support.
2. *Обеспечьте достаточный объем свободного пространства на диске.* Для работы потребуется не менее 50 Гбайт свободного пространства на диске при сборке образов[4].

[1]QuickEMUlator.
[2]Windows Subsystem for Linux - подсистема Windows для Linux.
[3]Опыт показывает, что YP достаточно эффективно работает на сборочном хосте с Linux Mageia 7.1.
[4]Опыт сборки образов для платформы HiFive Unleashed показывает объем исходных файлов и результатов сборки около 80 Гбайт. Дополнительно отметим, что каталог сборки лучше размещать на быстром диск (SAS или SSD), поскольку это значительно ускоряет работу.

3. *Выполните требования к версиям программ на хосте.* Система сборки OE может работать с любым современным дистрибутивом, включающим

   - Git 1.8.3.1 и выше;
   - tar 1.27 и выше;
   - Python 3.4.0 и выше.

   Если на сборочном хосте не выполняется какое-либо из этих требований, следует предпринять действия, указанные в разделе Required Git, tar, and Python Versions [3].

4. *Установка пакетов на хост разработки* сильно зависит от операционной системы и задач, которые планируется выполнять в YP. Если вы хотите работать со всеми классами, потребуется установка достаточно большого числа пакетов, как указано в разделе Required Packages for the Build Host [3].

После выполнения перечисленных действий система будет готова к использованию. Работа с рассмотрена в параграфе 2.4.1. Клонирование репозитория poky, использование расширяемого SDK[1] описано в разделе Using the Extensible SDK [2]. Работа с ядром Linux описана [7]. При использовании интерфейса управления Toaster следует обратиться к разделу Setting Up and Using Toaster [8].

2.2.2. Настройка CROPS

С CROPS[2] и контейнерами Docker можно создать среду разработки YP, независимую от операционной системы хоста. Можно организовать контейнер, который будет работать на машинах Windows, Mac, Linux для разработки в среде YP.

1. *Определение потребностей сборочного хоста.* Docker представляет собой платформу программных контейнеров, которую нужно установить на сборочном хосте. В зависимости от конкретного хоста можно установить разные программы для работы с контейнерами Docker. Информацию о поддерживаемых системах и установке программ для работы с контейнерами можно найти на странице Supported Platforms.
2. *Выбор устанавливаемых компонент.* В зависимости от соответствия сборочного хоста системным требованиям нужно установить пакет Docker CE Stable (в большинстве случаев) или Docker Toolbox.
3. *Переход на сайт установки для вашей платформы.* Воспользуйтесь ссылкой для выпуска Docker, соответствующего программной платформе сборочного хоста. Например, для Microsoft Windows версии 10 следует выбрать Docker CE Stable из списка поддерживаемых платформ.
4. *Установка программ.* После проверки соответствия установочного требованиям можно загрузить и установить программу, следуя инструкциям пакета установки.
   - Docker CE for Windows;
   - Docker CE for Macs;
   - Docker Toolbox for Windows;
   - Docker Toolbox for MacOS
   - Docker CE for CentOS;
   - Docker CE for Debian;
   - Docker CE for Fedora;
   - Docker CE for Ubuntu.
5. *Знакомство с системой Docker.* При необходимости ознакомьтесь с Docker и концепцией контейнеров на странице https://docs.docker.com/get-started/.
6. *Запуск Docker или Docker Toolbox* активизирует терминальное окно системы на вашем сборочном хосте.
7. *Настройка контейнеров для работы с YP.* На странице https://github.com/crops/docker-win-mac-docs/wiki приведены инструкции для работы на разных платформах (Linux, Mac или Windows). После выполнения установочных инструкций для вашей машины вы получите контейнеры Poky, eSDK и Toaster. Для знакомства с ними можно воспользоваться ссылками на указанной выше странице.

После установки контейнеров все готово для работы, как будто вы используете естественную машину Linux. Работа с контейнером Poky описана в параграфе 2.4.1. Клонирование репозитория poky. Описание работы с контейнером eSDK дано в разделе Using the Extensible SDK [2], а работа с системой Toaster в разделе Setting Up and Using Toaster [8].

2.3. Доступ к исходным файлам YP

В этом разделе описано как найти исходные файлы YP и работать с ними в проекте. Концепции и рекомендации по использованию Git с YP приведены в разделе Git [1]. Концепции работы с репозиторием YP описаны в разделе Yocto Project Source Repositories [1].

2.3.1. Доступ к репозиториям исходных кодов

Работа с копией репозитория исходных кодов является предпочтительным вариантом получения и использования YP. Список репозиториев YP можно найти на сайте http://git.yoctoproject.org. В частности, репозиторий poky размещается по ссылке http://git.yoctoproject.org/cgit/cgit.cgi/poky/. Процедура установки свежей версии описана ниже.

1. *Доступ к репозиториям.* Перейдите в браузере по ссылке http://git.yoctoproject.org с интерфейсом к репозиториям исходных кодов YP.

[1]Software Development Kit - комплект для разработки программ.
[2]CROssPlatformS - модель кросс-платформенной разработки с открытым исходным кодом.

2. *Выбор репозитория* по ссылке (например, poky).

3. *Определение URL для клонирования репозитория.* В нижней части страницы приведён идентификатор URL для клонирования репозитория (например, http://git.yoctoproject.org/poky). Процесс клонирования описан в параграфе 2.4.1. Клонирование репозитория poky.

2.3.2. Доступ к списку выпусков

YP поддерживает список выпусков (Index of Releases), где указаны файлы, включённые в YP. В отличие от репозиториев Git эти файлы являются архивами, соответствующими определённому времени. Рекомендуемым методом доступа к компонентам YP является клонирование репозитория Git и работа с локальной копией. Описанная в этом параграфе процедура относится к установке компонент из архивов.

1. *Доступ к списку выпусков.* Откройте браузер и перейдите по ссылке http://downloads.yoctoproject.org/releases для доступа к Index of Releases, где представлены компоненты выпуска (bitbake, sato и т. д.).

   Каталог yocto содержит архивы всех выпусков Poky, а каталог poky в Index of Releases применялся лишь для самых первых выпусков и сохранен только для полноты.

2. *Выберите компоненту* по ссылке в списке (например, yocto).

3. *Выберите архив* из списка выпусков. Например, щелчок по ссылке yocto-2.7.1 покажет файлы, связанные с выпуском YP 2.7.1 (например, poky-warrior-21.0.0.tar.bz2 для Poky).

4. *Загрузка архива.* Щёлкните по имени нужного архива для его загрузки.

2.3.3. Использование страницы загрузки

На сайте YP страница DOWNLOADS предназначена для выбора и загрузки архивов имеющихся выпусков YP. В отличие от репозиториев Git эти файлы представляют архив, сделанный в определённый момент, подобно файлам из Index of Releases, описанным в параграфе 2.3.2. Доступ к списку выпусков. Рекомендуемым методом доступа к компонентам YP является клонирование репозитория Git и работа с локальной копией. Описанная в этом параграфе процедура относится к установке компонент из архивов.

1. *Откройте сайт YP* в браузере по ссылке.

2. *Перейдите в область загрузки файлов,* выбрав опцию DOWNLOADS в раскрывающемся меню SOFTWARE наверху страницы.

3. *Выберите выпуск YP* в меню RELEASE (например, warrior, thud и т. д). Для сопоставления имён выпусков YP с номерами версий используйте страницу Releases. Можно воспользоваться ссылкой RELEASE ARCHIVE для получения меню выбора из всех выпусков YP.

4. *Загрузите инструменты или BSP* со страницы DOWNLOADS.

2.3.4. Доступ к «ночным сборкам»

YP поддерживает область «ночных сборок» в архивами на странице https://autobuilder.yocto.io//pub/nightly/. Эти архивы включают выпуски YP (poky), инструменты и сборки для поддерживаемых машин. Процедура получения ночных сборок описана ниже.

1. *Перейдите к списку ночных сборок,* открыв в браузере ссылку https://autobuilder.yocto.io//pub/nightly/.

2. *Выберите дату,* щёлкнув по нужной ссылке. Для получения последней сборки используйте CURRENT.

3. *Выберите сборку.* Например, для получения наиболее свежих инструментов служит ссылка toolchain.

4. *Выберите архив,* прокручивая список.

5. *Загрузите архив,* соответствующий выбранной дате и компоненте.

2.4. Клонирование и выбор ветвей

Для использования YP при разработке нужно установить выпуск пакета на хосте разработки. Этот набор локально установленных файлов называется в документации YP деревом (каталогом) исходных кодов (Source Directory).

Предпочтительным методом создания дерева кодов является использование Git для клонирования репозитория poky. Работа с клонированной копией репозитория позволяет включить результаты в YP или просто использовать последние версии программ из ветви разработки. Поскольку Git клонирует и поддерживает репозиторий с полной историей изменений, а работа выполняется с локальной копией репозитория, вы получаете доступ ко всем разрабатываемым ветвям YP и тегам, применяемым в репозитории.

2.4.1. Клонирование репозитория poky

Ниже перечислены этапы создания локальной копии репозитория Git poky.

1. *Организация локального каталога.* Создайте каталог для локального репозитория poky и перейдите в него.

2. *Клонирование репозитория.* Ниже приведён пример команды клонирования репозитория poky и её вывод.

```
$ git clone git://git.yoctoproject.org/poky
Cloning into 'poky'...
remote: Counting objects: 432160, done.
remote: Compressing objects: 100% (102056/102056), done.
remote: Total 432160 (delta 323116), reused 432037 (delta 323000)
Receiving objects: 100% (432160/432160), 153.81 MiB | 8.54 MiB/s, done.
Resolving deltas: 100% (323116/323116), done.
Checking connectivity... done.
```

Если не задана конкретная ветвь репозитория или тег, Git клонирует ветвь master в её текущем состоянии. Информация о выборе ветви или указании тега приведена в параграфах 2.4.2. Выбор ветви в Poky и 2.4.3. Выбор в Poky по тегу, соответственно.

После создания локального репозитория можно проверить его состояние. В примере показана 1 ветвь master.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
$ git branch
* master
```

Это говорит об идентичности локальной копии репозитория poky и его оригинала. При работе с локальной копией можно периодически использовать команду git pull --rebase для поддержки актуальности копии.

2.4.2. Выбор ветви в Poky

После клонирования репозитория poky вы получаете доступ ко всем его ветвям, каждая из которых уникальна и начинается от ветви master. Для просмотра и использования файлов той или иной ветви нужно узнать её имя, а затем переключиться (check out) на эту ветвь. Переключение на определённую ветвь по имени предоставляет доступ к текущему состоянию файлов этой ветви. Дальнейшая разработка будет выполняться применительно к файлам выбранной ветви.

1. *Переход в каталог Poky.* При наличии локальной копии репозитория poky Git, перейдите в соответствующий каталог. Создание локальной копии репозитория описано в параграфе 2.4.1. Клонирование репозитория poky.
2. *Определение ветвей репозитория.*

```
$ git branch -a
* master
  remotes/origin/1.1_M1
  remotes/origin/1.1_M2
  remotes/origin/1.1_M3
  remotes/origin/1.1_M4
  remotes/origin/1.2_M1
  remotes/origin/1.2_M2
  remotes/origin/1.2_M3
      ...
  remotes/origin/pyro
  remotes/origin/pyro-next
  remotes/origin/rocko
  remotes/origin/rocko-next
  remotes/origin/sumo
  remotes/origin/sumo-next
  remotes/origin/thud
  remotes/origin/thud-next
  remotes/origin/warrior
```

3. *Выбор ветви.* Выберите интересующую ветвь и перейдите в неё. Например, для доступа к файлам выпуска YP 2.7.1 (Warrior) используется приведённая ниже команда.

```
$ git checkout -b warrior origin/warrior
Branch warrior set up to track remote branch warrior from origin.
Switched to a new branch 'warrior'
```

Команда переключает ваш репозиторий на ветвь warrior и сообщает об отслеживании восходящей ветви origin/warrior.

Ниже приведена команда, отображающая ветви вашего текущего локального репозитория. Выбранная для работы ветвь помечена звёздочкой (*).

```
$ git branch
  master
* warrior
```

2.4.3. Выбор в Poky по тегу

Подобно ветвям, восходящий репозиторий использует теги для маркировки определённых фиксаций (commit), связанных со значимыми изменениями ветвей (например, точка официального выпуска). Можно связать локальный репозиторий с таким тегом, используя процедуру, похожую на выбор ветви. Переход на ветвь по тегу обеспечивает стабильный набор файлов, на которые не влияют изменения ветви, к которой относится этот тег.

1. *Переход в каталог Poky.* При наличии локальной копии репозитория poky Git, перейдите в соответствующий каталог. Создание локальной копии репозитория описано в параграфе 2.4.1. Клонирование репозитория poky.
2. *Получение тегов.* Для выбора ветви по имени тега нужно извлечь теги из восходящего репозитория в локальную копию с помощью приведённой ниже команды.

```
$ git fetch --tags
```

3. *Список имён тегов* с помощью команды git tag.

```
$ git tag
1.1_M1.final
1.1_M1.rc1
1.1_M1.rc2
1.1_M2.final
1.1_M2.rc1
   ...
yocto-2.5
yocto-2.5.1
```

```
yocto-2.5.2
yocto-2.5.3
yocto-2.6
yocto-2.6.1
yocto-2.6.2
yocto-2.7
yocto_1.5_M5.rc8
```

4. *Выбор ветви.*

```
$ git checkout tags/yocto-2.7.1 -b my_yocto_2.7.1
Switched to a new branch 'my_yocto_2.7.1'
$ git branch
  master
* my_yocto_2.7.1
```

Команда создаёт локальную ветвь my_yocto_2.7.1 (основанную на фиксации в восходящем репозитории с указанным тегом) и переключается на неё. В этом примере предоставляются файлы, связанные с выпуском YP 2.7.1 в ветви warrior.

Глава 3. Базовые задачи

В этой главе описаны фундаментальные процедуры, такие как создание уровней, добавление новых программных пакетов, перенос программ на новое оборудование (добавление машины) и т. п. Эти процедуры часто применяются в процессе работы с YP.

3.1. Уровни и их создание

Система сборки OE поддерживает организацию метаданных с множеством уровней (layer), позволяющих разделить разные типы настроек. Базовая информация об уровнях YP приведена в разделе The Yocto Project Layer Model [1].

3.1.1. Создание своего уровня

Создать свой уровень в OE очень просто. YP поставляется с инструментами для создания уровней. В этом параграфе описаны этапы создания уровней для лучшего понимания процесса. Средства для создания уровней описаны в разделе Creating a New BSP Layer Using the bitbake-layers Script [5] и параграфе 3.1.8. Создание базового уровня с помощью сценария bitbake-layers.

Ниже описаны этапы создания уровня без использования специальных инструментов

1. *Просмотр имеющихся уровней.* Перед созданием нового уровня нужно убедиться, что кто-либо уже не создал уровень с нужными вам метаданными. Можно просмотреть список OpenEmbedded Metadata Index, где указаны уровни, созданные сообществом OE, которые можно использовать в YP. Там может оказаться нужный или близкий к желаемому уровень.

2. *Создание каталога.* Создайте каталог для нового уровня в области, не связанной с каталогом исходных кодов YP (например, клонированным репозиторием poky).

   Хотя это не требуется, лучше использовать имена каталогов с префиксом meta-, например, meta-mylayer, meta-GUI_xyz, meta-mymachine.

   За редкими исключениями для имён уровней используют форму meta-root_name. Соблюдение этих соглашений об именах поможет предотвратить возникновение проблем, когда инструменты, компоненты или переменные «предполагают» имена уровней с префиксом meta-. Ярким примером служат файлы конфигурации, описанные на следующем этапе, где имена без префикса meta- добавляются в некоторые переменные, используемые в конфигурации.

3. *Создание файла конфигурации уровня.* В новом каталоге уровня нужно создать файл conf/layer.conf. Проще всего для этого взять файл конфигурации имеющегося уровня, скопировать его и внести требуемые изменения.

   Файл meta-yocto-bsp/conf/layer.conf в YP Source Repositories демонстрирует синтаксис конфигурационного файла. Следует заменить yoctobsp идентификатором своего уровня (например, machinexyz для уровня с именем meta-machinexyz).

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "yoctobsp"
BBFILE_PATTERN_yoctobsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_yoctobsp = "5"
LAYERVERSION_yoctobsp = "4"
LAYERSERIES_COMPAT_yoctobsp = "warrior"
```

- BBPATH - добавляет корневой каталог уровня в путь поиска BitBake. С помощью этой переменной BitBake находит файлы классов (.bbclass), конфигурационные файлы и файлы, включённые операторами include и require. BitBake в этих случаях использует первый файл, найденный по пути из переменной BBPATH. Это похоже на использование системной переменной PATH для поиска двоичных файлов. Поэтому рекомендуется применять для классов и конфигурации уникальные имена.
- BBFILES - определяет местоположение всех заданий уровня.

- BBFILE_COLLECTIONS - организует текущий уровень с помощью уникального идентификатора, используемого в системе сборки OE для указания этого уровня. В нашем примере идентификатор yoctobsp представляет контейнерный уровень meta-yocto-bsp.
- BBFILE_PATTERN - преобразуется в процессе разбора для представления имени каталога уровня.
- BBFILE_PRIORITY - задаёт приоритет для использования заданий уровня при поиске системой сборки OE заданий с одним именем из разных уровней.
- LAYERVERSION - задаёт номер версии для уровня. Этот номер можно применять для указания точной версии уровня в качестве зависимости при использовании переменной LAYERDEPENDS.
- LAYERSERIES_COMPAT - содержит список выпусков YP с которыми совместима данная версия. Это хороший способ указать, является ли ваш уровень текущим.

4. *Добавление содержимого* в зависимости от типа уровня. Если уровень включает поддержку машины, добавляется её конфигурация в каталог conf/machine/ внутри уровня. Если уровень добавляет правила distro указывается конфигурация дистрибутива в каталоге conf/distro/. При добавлении заданий их нужно помещать в подкаталоги recipes-* внутри уровня.

   Описание совместимой с YP иерархии уровней приведено в разделе Example Filesystem Layout [5].

5. *Необязательная проверка на совместимость.* Если вы хотите получить разрешение на использование логотипа YP Compatibility для уровня или приложения, использующего ваш уровень, нужно выполнить этот этап. Подробная информация приведена в параграфе 3.1.3. Подтверждение совместимости уровня с YP.

3.1.2. Рекомендации по организации уровней

Для создания уровней, которые будут просты в поддержке и не станут мешать другим уровням, следуйте приведённым ниже рекомендациям.

- *Избегайте полного перекрытия с заданиями из других уровней.* Избегайте копирования задания целиком на свой уровень с последующим изменением задания. Лучше использовать файл добавления (.bbappend) для переопределения лишь тех частей исходного задания, которые требуют этого.
- *Избегайте дублирования включаемых файлов.* Применяйте файлы добавления (.bbappend) для каждого задания, которое использует включаемый файл. Для нового задания, которому нужны включаемые файлы, используйте путь относительно каталога исходного уровня для ссылки на файл. Например, следует указывать require recipes-core/package/file.inc, а не просто require file.inc. Если возникает перекрытие включаемых файлов, это может говорить о недостатках включаемого файла на уровне, к которому он изначально относится. В таких случаях следует попытаться устранить недостаток, чтобы включаемые файлы не перекрывались. Например, можно попросить сопровождающего включаемый файл человека добавить в файл нужные переменные, чтобы упростить требуемые переопределения частей.
- *Структурируйте свои уровни.* Аккуратное использование переопределений в файлах добавления и размещение относящихся к конкретным машинам файлов внутри своего уровня может гарантировать предотвращения использования при сборке ошибочных метаданных и негативного влияния на сборку других машин. Ниже приведено несколько примеров.
  - *Изменяйте переменные для поддержки разных машин.* Предположим, что у вас есть уровень meta-one для сборок под машину one, применяется файл добавления base-files.bbappend и создана зависимость от foo в переменной DEPENDS = "foo"

    Зависимость создаётся в процессе любой сборки, включающей уровень meta-one. Однако эта зависимость может оказаться нужной не для всех машин. Например, предположим, что собирается образ для машины two и файл bblayers.conf включает уровень meta-one. В процессе сборки base-files для машины two будет добавлять зависимость от foo.

    Для того, чтобы изменения затрагивали только машину one используется переопределение в операторе DEPENDS_one = "foo".

    Аналогичный подход применяется при использовании операций _append и _prepend

    ```
    DEPENDS_append_one = " foo"
    DEPENDS_prepend_one = "foo "
    ```

    В качестве реального примера ниже приведена строка из задания для gnutls, добавляющая зависимость от argp-standalone при сборке с библиотекой musl C.

    ```
    DEPENDS_append_libc-musl = " argp-standalone"
    ```

    Избегайте операций += и =+, а также _append и _prepend.
  - *Размещайте файлы для конкретных машин в каталогах этих машин.* При наличии базового задания, такого как base-files.bb, которое содержит оператор SRC_URI, можно использовать файл добавления, заставляющий сборку использовать вашу версию файла. Например, файл добавление на вашем уровне meta-one/recipes-core/base-files/base-files.bbappend может расширять FILESPATH с использованием FILESEXTRAPATHS_prepend := "${THISDIR}/${BPN}:"

    Сборка для машины one возьмёт ваш файл для машины при его наличии в meta-one/recipes-core/base-files/base-files/. Однако, если сборка выполняется для другой машины и файл bblayers.conf включает уровень meta-one, а ваш файл, связанный с машиной, расположен в месте, которое задано первым в FILESPATH, сборка для всех машин будет использовать этот связанный с машиной файл.

    Можно обеспечить использование машинозависимого файла для конкретной машины путём его размещения в связанном с этой машиной каталоге. Например, вместо размещения в каталоге meta-one/recipes-core/base-files/base-files/, как показано выше, файл помещается в каталог

meta-one/recipes-core/base-files/base-files/one/. Это не только гарантирует использование файла при сборке для машины one, но и существенно ускоряет процесс сборки.

Таким образом, следует поместить все файлы, указанные в SRC_URI в связанный с машиной каталог внутри вашего уровня, чтобы использовать эти файлы только в сборках для данной машины.

- *Применение этапов YP Compatibility.* Если вы хотите получить право использовать логотип Yocto Project Compatibility со своим уровнем или приложением нужно выполнить действия, описанные в параграфе 3.1.3. Подтверждение совместимости уровня с YP.
- *Выполнение соглашений по именованию уровней.* Сохраняйте свои уровни в репозитории Git, используя формат meta-layer_name.
- *Группируйте свои уровни локально.* Клонируйте свой репозиторий вместе с другими мета-каталогами из дерева исходных кодов.

3.1.3. Подтверждение совместимости уровня с YP

При создании уровня для использования в YP полезно убедиться, что этот уровень хорошо взаимодействует с имеющимися уровнями YP (т. е. с уровнями, совместимыми с YP). Обеспечение совместимости упрощает использование уровня другими членами сообщества YP и позволяет использовать Yocto Project Compatible Logo. Знак Yocto Project Compatible Logo могут использовать только организации, являющиеся членами YP. Информация о членстве в YP представлена на сайте YP.

Программа совместимости YP включает процесс применения уровня, который запрашивает использование Yocto Project Compatibility Logo для уровня и приложений, состоящий из двух частей.

1. Успешное прохождение на уровне сценария (yocto-check-layer), проверяющего соблюдение ограничений, основанных на опыте использования уровней в реальном мире, и прохождение известных ловушек. Результат PASS в конце сценария говорит о совместимости уровня.
2. Заполнение заявки (https://www.yoctoproject.org/webform/yocto-project-compatible-registration).

Для получения права использовать логотип, нужно выполнить несколько условий:

- иметь возможность установить флажок, показывающий результат PASS при запуске сценария проверки;
- ответить Yes на вопросы заявки и предоставить приемлемое разъяснение для всех ответов No;
- быть членом Yocto Project Member Organization.

В оставшейся части этого раздела представлена информация о регистрационной форме и сценарии yocto-check-layer.

3.1.3.1. Применение программы Yocto Project Compatible

Используйте форму для подачи заявки на утверждение вашего уровня. После утверждения вы сможете указывать Yocto Project Compatibility Logo для уровня и использующих его приложений. При заполнении заявки следуйте инструкциям.

- *Контактные данные.* Предоставьте сведения для контактов с вами в требуемых полях. Укажите также версии выпусков YP, с которыми совместим ваш уровень.
- *Критерии согласия.* Укажите Yes или No для каждой записи в списке вопросов. В конце заявки имеется место для пояснений по всем пунктам, где вы ответили No.
- *Рекомендации.* Ответьте на вопросы, относящиеся к ядру Linux и результатам сборки.

3.1.3.2. Сценарий yocto-check-layer

Сценарий yocto-check-layer позволяет оценить совместимость вашего уровня с YP. Сценарием следует воспользоваться до подачи заявки, описанной в предыдущем параграфе. Для успешной обработки заявки нужно обеспечить получение результата PASS при работе сценария.

Сценарий включает 3 области проверки - COMMON, BSP и DISTRO. Например, для уровня дистрибутива (DISTRO) должны пройти тесты COMMON и DISTRO. Если ваш уровень является BSP, нужно пройти тесты COMMON и BSP.

Команды для запуска сценария приведены ниже.

```
$ source oe-init-build-env
$ yocto-check-layer your_layer_directory
```

В качестве your_layer_directory нужно указать реальное имя каталога с вашим уровнем.

Сценарий определяет тип уровня и запускает для него соответствующие тесты. Обзор тестов представлен ниже.

- *common.test_readme* проверяет наличие файла README на уровне и наличие содержимого в файле.
- *common.test_parse* проверяет возможность BitBake проанализировать файл без ошибок (bitbake -p).
- *common.test_show_environment* проверяет наличие ошибок в глобальном окружении и среде задания (bitbake -e).
- *common.test_signatures* проверяет отсутствие на уровнях BSP и DISTRO заданий, меняющих подписи.
- *bsp.test_bsp_defines_machines* проверяет наличие конфигураций для машин на уровне BSP.
- *bsp.test_bsp_no_set_machine* проверяет, что уровень BSP не задаёт машину при его добавлении.
- *distro.test_distro_defines_distros* проверяет наличие конфигураций на уровне DISTRO.
- *distro.test_distro_no_set_distro* проверяет, что уровень DISTRO не задаёт дистрибутив при своём добавлении.

3.1.4. Включение уровня

Перед началом использования уровня системой сборки OE этот уровень нужно включить в конфигурацию, для чего просто добавляется путь к этому уровню в переменную BBLAYERS файла conf/bblayers.conf, хранящегося в каталоге сборки[1]. Ниже приведён пример добавления уровня meta-mylayer.

```
# POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
# changes incompatibly
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/user/poky/meta \
  /home/user/poky/meta-poky \
  /home/user/poky/meta-yocto-bsp \
  /home/user/poky/meta-mylayer \
  "
```

BitBake анализирует каждый файл conf/layer.conf сверху вниз в соответствии с переменной BBLAYERS в файле conf/bblayers.conf. При обработке каждого файла conf/layer.conf программа BitBake добавляет задания, классы и конфигурации, содержащиеся в указанном уровне.

3.1.5. Использование файлов .bbappend на уровне

Задание, добавляющее метаданные в другое задание, называется в BitBake файлом добавления. Эти файлы используют расширение .bbappend, тогда как файлы, в которые метаданные добавляются, - расширение .bb.

Можно применять файл .bbappend на своём уровне или менять содержимое задания на другом уровне без копирования таких заданий на свой уровень. Файлы .bbappend размещаются на вашем уровне, тогда как файл .bb, в который добавляются метаданные, может размещаться на другом уровне.

Возможность добавлять информацию в имеющиеся задания не только избавляет от дублирования, но и автоматически переносит изменения заданий других уровней на ваш уровень. Если бы вы копировали эти задания, пришлось бы вручную вносить изменения.

При создании файла добавления нужно использовать корневое имя соответствующего файла задания. Например, файл добавления someapp_2.7.1.bbappend должен применяться к файлу someapp_2.7.1.bb. Это означает зависимость исходного задания и файла добавления от номера версии. Если задание переименовано с учётом новой версии, нужно будет обновить и, возможно, переименовать соответствующий файл .bbappend. В процессе сборки BitBake выводит сообщения об ошибках при запуске, если обнаруживается, что для файла .bbappend нет задания с соответствующим именем. Обработка таких ошибок рассмотрена в описании переменной BB_DANGLINGAPPENDS_WARNONLY.

Рассмотрим в качестве примера задание formfactor и соответствующий файл добавления formfactor, хранящиеся в дереве исходных кодов. Ниже показан основной файл задания formfactor_0.0.bb, хранящийся на уровне meta в каталоге meta/recipes-bsp/formfactor.

```
SUMMARY = "Device formfactor information"
SECTION = "base"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;md5=3da9cfbcb788c80a0384361b4de20420"
PR = "r45"

SRC_URI = "file://config file://machconfig"
S = "${WORKDIR}"

PACKAGE_ARCH = "${MACHINE_ARCH}"
INHIBIT_DEFAULT_DEPS = "1"

do_install() {
        # Install file only if it has contents
        install -d ${D}${sysconfdir}/formfactor/
        install -m 0644 ${S}/config ${D}${sysconfdir}/formfactor/
        if [ -s "${S}/machconfig" ]; then
    install -m 0644 ${S}/machconfig ${D}${sysconfdir}/formfactor/
        fi
}
```

Отметим в этом файле переменную SRC_URI, которая указывает системе сборки OE места поиска файлов при сборке.

Ниже показано содержимое файла добавления formfactor_0.0.bbappend с уровня Raspberry Pi BSP Layer называющегося meta-raspberrypi. Файл хранится на этом уровне в каталоге recipes-bsp/formfactor.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
```

По умолчанию система сборки применяет для поиска файлов переменную FILESPATH. Этот файл добавления расширяет сферу поиска путём установки переменной FILESEXTRAPATHS. Назначение этой переменной в файле .bbappend является наиболее надёжным и рекомендуемым методом добавления каталогов в путь поиска при сборке.

Оператор в этом примере расширяет переменную путём добавления в конце значения ${THISDIR}/${PN}, преобразуемого в каталог formfactor в том же каталоге, где хранится файл добавления (meta-raspberrypi/recipes-bsp/formfactor). Это предполагает наличие структуры каталогов поддержки для файлов и исправлений (patch), применяемых уровнем.

[1]В реальности добавление уровня путём редактирования файла (по крайней мере для bitbake версии 1.44.0) результата не даёт и уровень не виден в системе сборки. Эффективно удалось добавить уровень лишь командой bitbake-layers add-layer из каталога build.

Использование оператора непосредственного преобразования := в этом случае важно, поскольку указывается текущий каталог THISDIR. Двоеточие в конце обеспечивает отделение последующих элементов.

BitBake автоматически определяет значение переменной THISDIR и вам не следует устанавливать его. Использование _prepend с переменной FILESEXTRAPATHS обеспечивает добавление пути поиска в начало имеющегося списка.

Не все файлы .bbappend добавляют новые файлы, многие просто добавляют опции сборки (например, systemd). В таких случаях файл добавления не будет включать оператора FILESEXTRAPATHS.

3.1.6. Приоритизация уровня

Каждый уровень имеет определённый приоритет, который определяет предпочтительность уровня при наличии одноимённых файлов заданий на нескольких уровнях. В таких случаях берётся задание уровня с высшим приоритетом. Значения приоритета влияют также на порядок применения файлов .bbappend к одному заданию. Приоритет можно задать вручную или позволить рассчитать его системе сборки на основе зависимостей между уровнями.

Для задания приоритета вручную служит переменная BBFILE_PRIORITY с добавлением корневого файла уровня.

```
BBFILE_PRIORITY_mylayer = "1"
```

Возможны ситуации, когда задание с меньшим номером версии PV на уровне будет иметь более высокий приоритет.

Уровень приоритета не влияет на порядок предпочтения файлов .conf и .bbclass, но это может появиться в будущих версиях BitBake.

3.1.7. Управление уровнями

Можно использовать инструмент управления уровнями bitbake-layers для просмотра структуры заданий в проекте с множеством уровней. Возможность увидеть отчёт о настроенных уровнях с их путями и приоритетами, файлами .bbappend и заданиями, к которым они относятся, может оказать помощь при поиске неполадок.

Информацию о работе с инструментом можно получить по команде

```
$ bitbake-layers --help
NOTE: Starting bitbake server...
usage: bitbake-layers [-d] [-q] [-F] [--color COLOR] [-h] <subcommand> ...

BitBake layers utility

optional arguments:
  -d, --debug           Enable debug output
  -q, --quiet           Print only errors
  -F, --force           Force add without recipe parse verification
  --color COLOR         Colorize output (where COLOR is auto, always, never)
  -h, --helpshow this help message and exit

subcommands:
  <subcommand>
    show-layers         show current configured layers.
    show-overlayed      list overlayed recipes (where the same recipe exists
                            in another layer)
    show-recipes        list available recipes, showing the layer they are
                            provided by
    show-appends        list bbappend files and recipe files they apply to
    show-cross-depends  Show dependencies between recipes that cross layer
                            boundaries.
    add-layer           Add one or more layers to bblayers.conf.
    remove-layer        Remove one or more layers from bblayers.conf.
                          flatten flatten layer configuration into a separate output
                            directory.
    layerindex-fetch    Fetches a layer from a layer index along with its
                            dependent layers, and adds them to conf/bblayers.conf.
    layerindex-show-depends
                            Find layer dependencies from layer index.
    create-layer        Create a basic layer

  Use bitbake-layers <subcommand> --help to get help on a specific command
```

Краткие описания команд приведены ниже.

help
Выводит общую справку или информацию о конкретной команде.

show-layers
Показывает настроенные в данный момент уровни.

show-overlayed
Показывает перекрывающиеся задания, т. е. одноимённые задания на разных уровнях.

show-recipes
Показывает задания и уровни, к которым они относятся.

show-appends
Показывает файлы .bbappend и задания, к которым они относятся.

show-cross-depends
Перечисляет зависимости между заданиями разных уровней.

add-layer
Добавляет уровень в файл bblayers.conf.

remove-layer
Удаляет уровень из файла bblayers.conf.

flatten

Собирает конфигурацию уровня с отдельный выходной каталог. При сборке конфигурации создаётся «плоский» каталог с содержимым всех уровней и удалением всех перекрывающихся заданий и файлов .bbappend для них. Возможно после этого потребуется некоторая очистка вручную, как описано ниже.

- Не являющиеся заданиями файлы (например, исправления) переписываются и команда flatten выдаёт для них предупреждения.
- Все, что выходит за пределы обычной установки уровня, добавляется в файл layer.conf. Используется лишь файл layer.conf уровня с наименьшим приоритетом.
- Переписанные и добавленные элементы из файлов .bbappend должны быть очищены. Содержимое каждого файла .bbappend в конечном итоге попадает в «плоское» задание. Однако при наличии добавленных или изменённых значений переменных потребуется самостоятельно привести их в порядок. Рассмотрим пример, где команда bitbake-layers добавляет строку #### bbappended ... (чтобы понять, откуда берутся следующие строки).

```
...
DESCRIPTION = "A useful utility"
...
EXTRA_OECONF = "--enable-something"
...

#### bbappended from meta-anotherlayer ####

DESCRIPTION = "Customized utility"
EXTRA_OECONF += "--enable-somethingelse"
```

В идеальном случае это следует привести к виду, показанному ниже.

```
...
DESCRIPTION = "Customized utility"
...
EXTRA_OECONF = "--enable-something --enable-somethingelse"
...
```

layerindex-fetch

Извлекает уровень из списка уровней вместе с зависимыми уровнями и добавляет их в файл conf/bblayers.conf.

layerindex-show-depends

Определяет зависимости уровня из списка уровней.

create-layer

Создаёт базовый уровень.

3.1.8. Создание базового уровня с помощью сценария bitbake-layers

Сценарий bitbake-layers с параметром create-layer упрощает создание нового уровня общего назначения. Для использования уровня в системе сборки OE нужно добавить этот уровень в конфигурационный файл bblayers.conf (см. 3.1.9. Добавление уровня с помощью сценария bitbake-layers). По умолчанию bitbake-layers create-layer создаёт уровень со следующими параметрами:

- приоритет уровня 6;
- каталог conf содержит файл layer.conf;
- создаётся каталог recipes-example с подкаталогом example, содержащим файл задания example.bb;
- файл COPYING.MIT с лицензией (сценарий предполагает использование лицензии MIT, типичной для большинства уровней), относящейся к содержимому уровня;
- файл README с описанием содержимого нового уровня.

В простейшем случае для создания уровня можно использовать приведённую ниже команду, которая создаёт в текущем каталоге уровень, имя которого соответствует параметру your_layer_name.

```
$ bitbake-layers create-layer your_layer_name
```

Например, приведённая ниже команда создаст уровень meta-scottrif в домашнем каталоге.

```
$ cd /usr/home
$ bitbake-layers create-layer meta-scottrif
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer meta-scottrif'
```

Если нужно установить для уровня приоритет, отличный от 6, можно использовать опцию --priority или отредактировать значение BBFILE_PRIORITY в файле conf/layer.conf после его создания сценарием. Если нужно изменить имя примера файла задания, его можно задать с помощью опции --example-recipe-name.

Простейшим способом разобраться с командой bitbake-layers create-layer является её использование. Для получения справки о работе со сценарием введите команду

```
$ bitbake-layers create-layer --help
NOTE: Starting bitbake server...
usage: bitbake-layers create-layer [-h] [--priority PRIORITY]
[--example-recipe-name EXAMPLERECIPE]
layerdir

Create a basic layer

positional arguments:
  layerdir  Layer directory to create

optional arguments:
  -h, --helpshow this help message and exit
  --priority PRIORITY, -p PRIORITY
Layer directory to create
```

```
      --example-recipe-name EXAMPLERECIPE, -e EXAMPLERECIPE
    Filename of the example recipe
```

3.1.9. Добавление уровня с помощью сценария bitbake-layers

Создав уровень, нужно добавить его в свой файл bblayers.conf. Это позволяет системе сборки OE узнать об этом уровне, чтобы искать в нем метаданные. Для добавления уровня служит команда bitbake-layers add-layer your_layer_name.

Ниже приведён пример добавления уровня meta-scottrif в файл конфигурации. После команды добавления уровня использована команда, показывающая уровни из вашего файла bblayers.conf.

```
$ bitbake-layers add-layer meta-scottrif
NOTE: Starting bitbake server...
Parsing recipes: 100% |##########################################################| Time: 0:00:49
Parsing of 1441 .bb files complete (0 cached, 1441 parsed). 2055 targets, 56 skipped, 0 masked, 0 errors.

$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer     path  priority
==========================================================================
meta           /home/scottrif/poky/meta 5
meta-poky      /home/scottrif/poky/meta-poky 5
meta-yocto-bsp /home/scottrif/poky/meta-yocto-bsp 5
workspace      /home/scottrif/poky/build/workspace 99
meta-scottrif  /home/scottrif/poky/build/meta-scottrif 6
```

Добавление уровня в этот файл позволяет системе сборки найти уровень в процессе сборки. Система сборки просматривает уровни от начала списка к концу.

3.2. Настройка образов

Образы можно настраивать с учётом реальных требований, как описано в этом параграфе.

3.2.1. Настройка с помощью local.conf

Одним из простейших способов настройки образа является добавление пакетов через файл local.conf. Поскольку область действия этого файла ограничена, метод обычно позволяет лишь добавлять пакеты и недостаточно гибок для создания своего настраиваемого образа. При добавлении пакетов с использованием локальных переменных нужно отдавать себе отчёт в том, что эти изменения влияют на каждую сборку и, следовательно, на все создаваемые образы.

Для добавления пакета с использованием локального файла конфигурации служит переменная IMAGE_INSTALL с оператором _append.

```
    IMAGE_INSTALL_append = " strace"
```

Важно соблюдать показанный в примере синтаксис, в частности пробел перед именем добавляемого пакета, поскольку оператор _append пробелы (разделители) не добавляет. Кроме того, нужно использовать _append, а не оператор +=, если вы не хотите столкнуться с проблемой упорядочения. Показанный вариант безоговорочно добавляет значение в конец имеющегося списка, что позволяет предотвратить связанные с упорядочением проблемы в результате использования переменной в образах и файлах заданий с операторами вида ?=.

Как было отмечено, IMAGE_INSTALL_append влияет на все образы, но можно расширить синтаксис, задавая применение переменной лишь для конкретного образа в форме IMAGE_INSTALL_append_pn-core-image-minimal = " strace".

Можно добавлять пакеты похожим способом через переменную CORE_IMAGE_EXTRA_INSTALL, которая воздействует только на образы core-image-*.

3.2.2. Настройка с помощью IMAGE_FEATURES и EXTRA_IMAGE_FEATURES

Другим вариантом является использование переменных IMAGE_FEATURES и EXTRA_IMAGE_FEATURES. Эти переменные близки по назначению, лучше пользоваться IMAGE_FEATURES из задания и EXTRA_IMAGE_FEATURES из файла local.conf в каталоге сборки.

Разобраться с настройками поможет файл meta/classes/core-image.bbclass. Этот класс указывает доступные свойства IMAGE_FEATURES, большинство которых отображается на группы пакетов, а некоторые (например, debug-tweaks и read-only-rootfs) задают общие настройки конфигурации.

Содержимое IMAGE_FEATURES просматривается а затем свойства отображаются или настраиваются соответствующим образом. На основе этой информации система сборки автоматически добавляет нужные пакеты и конфигурации в переменную IMAGE_INSTALL. Фактически это включает дополнительные функции путём расширения классов или создания собственного класса для использования с файлами образов .bb.

Следует использовать переменную EXTRA_IMAGE_FEATURES из локального файла конфигурации. Использование отдельной области, откуда можно управлять свойствами через эту переменную поможет избежать переопределения свойств в задании для образа, которые указаны в переменной IMAGE_FEATURES. Значение переменной EXTRA_IMAGE_FEATURES добавляется к IMAGE_FEATURES в файле meta/conf/bitbake.conf.

Для иллюстрации этих возможностей рассмотрим пример выбора сервера SSH. YP включает два сервера SSH - Dropbear и OpenSSH. Первый является минимальным сервером SSH подходящим для сред с ограниченными ресурсами, а OpenSSH является широко распространенным стандартным сервером SSH. По умолчанию для образа core-image-sato настроено использование Dropbear, а образы core-image-full-cmdline и core-image-lsb включают OpenSSH. Образ core-image-minimal не поддерживает сервер SSH. Можно настроить свой образ и изменить принятые по умолчанию значения. Для этого следует изменить переменную IMAGE_FEATURES или EXTRA_IMAGE_FEATURES в файле local.conf, указав пакет ssh-server-dropbear или ssh-server-openssh. Полный список свойств образов, включённых в YP, приведён в разделе Images [3].

3.2.3. Настройка с помощью файлов .bb

Можно также настроить образ, создав своё задание, определяющее дополнительные программы для образа

```
IMAGE_INSTALL = "packagegroup-core-x11-base package1 package2"

inherit core-image
```

Указание программ через пользовательское задание обеспечивает полный контроль содержимого образа. Важно указывать корректные имена пакетов в переменной IMAGE_INSTALL, используя нотацию OE, а не (например, glibc-dev, а не libc6-dev).

Другим вариантом создания своего образа служит копирование и изменение имеющегося образа. Например, можно создать образ на основе core-image-sato с добавлением пакета strace, скопировав файл meta/recipes-sato/images/core-image-sato.bb и добавив в конце копии строку IMAGE_INSTALL += "strace".

3.2.4. Настройка с помощью пользовательских групп пакетов

Для комплексных образов лучшим способом настройки является создание группы заданий, которая будет применяться при сборке образов. Примером такой группы служит meta/recipes-core/packagegroups/packagegroup-base.bb. В этом задании переменная PACKAGES указывает группы создаваемых пакетов. Оператор inherit packagegroup задаёт принятые по умолчанию значения и автоматически добавляет пакеты -dev, -dbg и -ptest для каждого пакета, указанного в PACKAGES. Пакеты с наследованием следует размещать в верхней части задания и обязательно до оператора PACKAGES. Для каждого пакета из PACKAGES можно использовать записи RDEPENDS и RRECOMMENDS для предоставления списков пакетов, которые следует включать в родительский пакет. Примеры этого имеются в задании packagegroup-base.bb.

Ниже приведён краткий пример, иллюстрирующий основные части.

```
DESCRIPTION = "My Custom Package Groups"

inherit packagegroup

PACKAGES = "\
    packagegroup-custom-apps \
    packagegroup-custom-tools \
    "

RDEPENDS_packagegroup-custom-apps = "\
    dropbear \
    portmap \
    psplash"

RDEPENDS_packagegroup-custom-tools = "\
    oprofile \
    oprofileui-server \
    lttng-tools"

RRECOMMENDS_packagegroup-custom-tools = "\
    kernel-module-oprofile"
```

В этом примере создаётся две группы пакетов с указанием действительных и рекомендуемых зависимостей - packagegroup-custom-apps и packagegroup-custom-tools. Для сборки образа с использованием этой группы пакетов нужно добавить packagegroup-custom-apps и/или packagegroup-custom-tools в переменную IMAGE_INSTALL.

3.2.5. Настройка имени хоста для образа

По умолчанию настроенной имя хоста (/etc/hostname) в образе совпадает с именем машины. Например, для MACHINE = qemux86, настроенным именем хоста в файле /etc/hostname будет qemux86.

Можно изменить это имя, указав его в переменной hostname в задании base-files с помощью файла добавления или конфигурационного файла. Ниже приведён пример с файлом добавления.

```
hostname="myhostname"
```

В файле конфигурации замена будет иметь вид hostname_pn-base-files = "myhostname".

Смена принятого по умолчанию значения hostname может быть полезна в нескольких случаях. Например, может потребоваться расширенное тестирования образа, где нужно будет отличать тестируемые варианты от типового. В таком случае можно сменить имя хоста, например, на testme. Если переменная не установлена, у образа не будет заданного по умолчанию в файловой системе имени, как показано ниже.

```
hostname_pn-base-files = ""
```

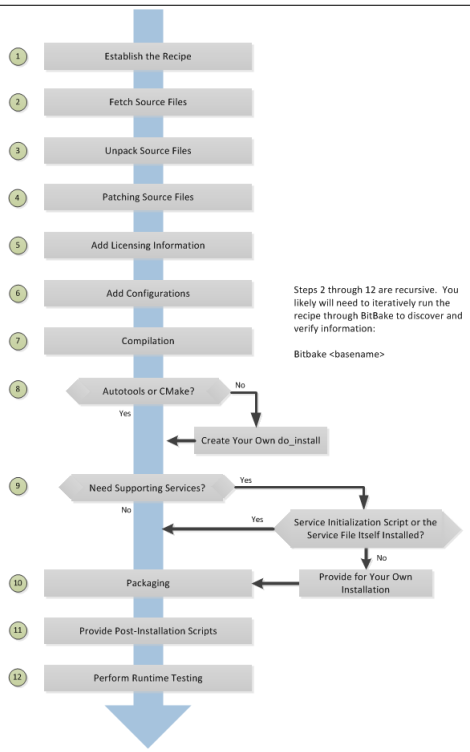
Отсутствие заданного по умолчанию имени удобно в средах с динамическим назначением имён хостов.

3.3. Создание новых заданий

Задания (файлы .bb) являются важнейшими компонентами среды YP. Каждая программная компонента создаётся системой сборки OE на основе определяющего её задания. В этом разделе описаны все аспекты работы с заданиями. Дополнительную информацию о работе с заданиями можно найти в разделе Required [3].

3.3.1. Обзор

На рисунке показан базовый процесс создания нового задания.



3.3.2. Нахождение или автоматическое создание базового задания

Всегда можно создать новое задание «с нуля», однако есть 3 варианта, ускоряющие подготовку новых заданий:

- *devtool add* - команда, помогающая создать задание и среду разработки;
- *recipetool create* - команда YP для автоматизированного создания заданий на основе файлов исходного кода;
- *имеющиеся задания* можно скопировать и изменить в соответствии с задачами.

Синтаксис заданий описан в параграфе 3.3.23. Синтаксис заданий.

3.3.2.1. Создание с помощью devtool add

Команда devtool add использует такую же логику автоматического создания заданий, как описанная ниже команда recipetool create. Дополнительно devtool add организует окружение, упрощающее правку (patch) исходных кодов и внесение изменений в задания, которые часто требуются при добавлении заданий для сборки новой части кода. Полное описание команды devtool add приведено в разделе A Closer Look at devtool add [2].

3.3.2.2. Создание с помощью recipetool create

Команда recipetool create автоматизирует создание базового задания на основе набора файлов с исходным кодом. Когда можно извлечь или указать исходные файлы, команда будет создавать задание и автоматически настраивать все параметры сборки в нем. Например, приложение может создаваться с помощью autotools и подготовка базового задания с помощью recipetool приведёт к включению в задание зависимостей, лицензионных требований и контрольных сумм.

Запуск команды выполняется из каталога сборки после выполнения сценария настройки окружения (oe-init-build-env). Для получения справки о работе с инструментом служит команда

```
$ recipetool -h
NOTE: Starting bitbake server...
usage: recipetool [-d] [-q] [--color COLOR] [-h] <subcommand> ...

OpenEmbedded recipe tool

options:
  -d, --debug     Enable debug output
  -q, --quiet     Print only errors
  --color COLOR   Colorize output (where COLOR is auto, always, never)
  -h, --help      show this help message and exit

subcommands:
  create          Create a new recipe
  newappend       Create a bbappend for the specified target in the specified
                    layer
  setvar          Set a variable within a recipe
  appendfile      Create/update a bbappend to replace a target file
  appendsrcfiles  Create/update a bbappend to add or replace source files
  appendsrcfile   Create/update a bbappend to add or replace a source file
Use recipetool <subcommand> --help to get help on a specific command
```

Команда recipetool create -o OUTFILE создаёт базовое задание с именем OUTFILE и корректно размещает его на уровне, содержащем исходные файлы.

Ниже приведён синтаксис для подготовки на базе кодов source задания, размещаемого на уровне исходного кода.

```
recipetool create -o OUTFILE source
```

Следующая команда создаёт задание с использованием кода, извлекаемого из source. Этот код помещается на свой уровень, указанный EXTERNALSRC.

```
recipetool create -o OUTFILE -x EXTERNALSRC source
```

Ещё одна команда создаёт задание на основе source, указывая recipetool необходимость создания отладочной информации. Задание размещается в имеющемся уровне исходных кодов.

```
recipetool create -d -o OUTFILE source
```

3.3.2.3. Нахождение и использование похожего задания

Перед созданием нового задания зачастую полезно ознакомиться с имеющимися и выбрать среди них подходящее в качестве основы. Сообщества YP и OE поддерживают множество заданий, среди которых наверняка найдётся подходящий образец. Список имеющихся заданий доступен по ссылке OpenEmbedded Layer Index.

Работа с имеющимся заданием и шаблоном служит логичным началом процесса. Ниже рассмотрены особенности обоих методов.

- *Поиск и изменение похожего задания* подходит, когда вы хорошо знакомы с имеющимися заданиями, но мало полезен для новичков. Некоторый риск при использовании данного метода связан с возможным наличием в выбранном задании области, которая совсем не связана с вашими задачами и может потребоваться её создание с нуля. Однако все такие риски связаны с недостаточным знакомством с имеющимися заданиями.
- *Использование шаблона задания.* Если по какой-то причине вы не хотите использовать recipetool и не нашли подходящего в качестве образца задания, можно воспользоваться приведённой ниже структурой, включающей основные разделы нового задания.

```
DESCRIPTION = ""
HOMEPAGE = ""
LICENSE = ""
SECTION = ""
DEPENDS = ""
LIC_FILES_CHKSUM = ""

SRC_URI = ""
```

3.3.3. Сохранение и именование заданий

Базовое задание следует поместить на свой уровень и дать ему подходящее имя. Корректное размещение заданий позволит системе сборки OE находить их при использовании BitBake для обработки.

- *Сохранение заданий.* Система сборки OE находит задания с помощью файла conf/layer.conf на вашем уровне и переменной BBFILES. Эта переменная задаёт путь поиска заданий для системы сборки, как показано ниже.

  ```
  BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
  ${LAYERDIR}/recipes-*/*/*.bbappend"
  ```

  Поэтому следует сохранять задания внутри своего уровня, чтобы их можно было найти. Информация о структуре уровней приведена в разделе 3.1. Уровни и их создание.
- *Именование заданий.* При выборе имён для заданий используйте формат basename_version.bb.

  Используйте символы нижнего регистра и не включайте в имена зарезервированные суффиксы -native, -cross, -initial, -dev, пока это не обусловлено применением задания. Ниже приведено несколько примеров.

  ```
  cups_1.7.0.bb
  gawk_4.0.2.bb
  irssi_0.8.16-rc1.bb
  ```

3.3.4. Запуск сборки задания

Подготовка нового задания обычно происходит в несколько итераций, требующих использования BitBake для многократной обработки задания с целью нахождения и добавления информации в файл задания. Предположим, что вы воспользовались сценарием настройки среды сборки (oe-init-build-env) и текущим является сборочный каталог, а для обработки задания применяется BitBake. Для запуска достаточно указать базовое имя задания (basename).

```
$ bitbake basename
```

В процессе работы система сборки OE создаёт временный рабочий каталог для каждого задания (${WORKDIR}), в котором хранятся извлечённые файлы исходного кода, журналы работы (log), промежуточные файлы компиляции, файлы пакетов и т. п. Путь к временному рабочему каталогу каждого задания зависит от рабочего контекста. Проще всего узнать этот путь с помощью команды bitbake -e basename | grep ^WORKDIR=.

Предположим, например, что каталогом верхнего уровня для исходных кодов служит poky, сборочным каталогом - poky/build, а целевой системой является qemux86-poky-linux. Пусть задание называется foo_1.3.0.bb. В этом случае рабочим каталогом для сборки пакета будет poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0.

Внутри этого каталога можно найти такие поддкаталоги, как image, packages-split и temp. После завершения сборки можно рассмотреть их содержимое для оценки результатов сборки. Журналы сборки для каждого задания можно найти в каталоге temp этого задания (например, poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0/temp). Файлы журналов будут названы log.taskname (например, log.do_configure, log.do_fetch, log.do_compile).

Дополнительная информация о процессе сборки приведена в разделе The Yocto Project Development Environment [1].

3.3.5. Извлечение кода

Первым делом задание должно указать способ извлечения исходных файлов. Выборка исходного кода определяется в основном переменной SRC_URI. Задание должно включать переменную SRC_URI, указывающую место размещения исходных кодов. Графическое представление мест размещения исходных кодов дано в разделе Sources [1].

Задача do_fetch использует префикс каждого элемента переменной SRC_URI для определения сборщика, применяемого для получения исходных кодов. Задача do_patch использует эту переменную после извлечения исходных кодов для применения правок (patch). Система сборки применяет FILESOVERRIDES для сканирования каталогов с локальными файлами, указанных в SRC_URI.

Переменная SRC_URI в задании должна указывать каждое уникальное место размещения исходного кода. Рекомендуется не задавать жёстко пути в URL, указанных в SRC_URI, а пользоваться переменной ${PV}, которая сообщает сборщику версию, указанную именем файла задания. Такое указание версии упрощает обновления задания , сводя его к простому переименованию файла в соответствии с новой версией. Ниже приведён пример из задания meta/recipes-devtools/cdrtools/cdrtools-native_3.01a20.bb, содержащего исходные файлы из архива.

```
SRC_URI = "ftp://ftp.berlios.de/pub/cdrecord/alpha/cdrtools-${PV}.tar.bz2"
```

Файлы, указанные в SRC_URI, имена которых имеют типовое расширение (например, .tar, .tar.gz, .tar.bz2, .zip), извлекаются автоматически задачей do_unpack. Другой пример указания файлов приведён в параграфе 3.3.21.2. Пакеты с автоматической настройкой.

Другим способом указания исходных файлов служит SCM. Для репозиториев Git нужно задать SRCREV и следует включить в PV номер выпуска с помощью SRCPV, как показано ниже для meta/recipes-kernel/blktrace/blktrace_git.bb

```
SRCREV = "d6918c8832793b4205ed3bfede78c2f915c23385"

PR = "r6"
PV = "1.0.5+git${SRCPV}"

SRC_URI = "git://git.kernel.dk/blktrace.git \
           file://ldflags.patch"
```

Если SRC_URI содержит URL, указывающие отдельные файлы, извлечённые с удалённого сервера, который не является системой контроля версий, BitBake пытается проверить файлы по контрольным суммам из задания, чтобы убедиться в отсутствии изменений в файлах с момента подготовки задания. Применяются контрольные суммы SRC_URI[md5sum] и SRC_URI[sha256sum]. Если переменная SRC_URI включает несколько URL (кроме SCM URL), нужно обеспечить проверку md5 и sha256 для каждого URL. В таких случаях подставляется каждый URL как часть SRC_URI, а затем используется при проверке контрольных сумм, как показано ниже

```
SRC_URI = "${DEBIAN_MIRROR}/main/a/apmd/apmd_3.2.2.orig.tar.gz;name=tarball \
${DEBIAN_MIRROR}/main/a/apmd/apmd_${PV}.diff.gz;name=patch"

SRC_URI[tarball.md5sum] = "b1e6309e8331e0f4e6efd311c2d97fa8"
SRC_URI[tarball.sha256sum] = "7f7d9f60b7766b852881d40b8ff91d8e39fccb0d1d913102a5c75a2dbb52332d"

SRC_URI[patch.md5sum] = "57e1b689264ea80f78353519eece0c92"
SRC_URI[patch.sha256sum] = "7905ff96be93d725544d0040e425c42f9c05580db3c272f11cff75b9aa89d430"
```

Корректные значения md5 и sha256 могут быть доступны вместе с другими подписями (md5, sha1, sha256, GPG и т. п.) на странице загрузки исходных файлов. Поскольку система сборки OE поддерживает только sha256sum и md5sum, другие контрольные суммы придётся проверять вручную.

Если при попытке сборки задания контрольные суммы SRC_URI не указаны или отличаются от указанных, система сборки будет выдавать ошибку для каждой пропущенной или неверной контрольной суммы. В сообщениях будет указываться реальная контрольная сумма файла. После обеспечения корректных контрольных сумм можно скопировать их в задание и запустить сборку снова.

Заключительный пример слегка взят из задания unicode/rxvt-unicode_9.20.bb. Оператор SRC_URI указывает множество файлов для задания - архивы, исправления, файлы рабочего стола и пиктограммы.

```
SRC_URI = "http://dist.schmorp.de/rxvt-unicode/Attic/rxvt-unicode-${PV}.tar.bz2 \
           file://xwc.patch \
           file://rxvt.desktop \
           file://rxvt.png"
```

При указании локальных файлов с помощью file:// URI система сборки извлекает файлы с локальной машины. Пути задаются относительно переменной FILESPATH и поиск выполняется в каталогах в порядке ${BP}, ${BPN} и FILES. Каталоги предполагаются расположенными в каталоге, где размещается файл задания или добавления. Пример поиска приведён в параграфе 3.3.21.1. Пакет с одним файлом .c.

В первом примере был указан patch-файл. Такие файлы обычно имеют расширение .patch или .diff, но могут применять и суффиксы сжатых архивов, такие как diff.gz или patch.bz2. Система сборки будет автоматически применять исправления из таких файлов, как описано в параграфе 3.3.7. Применение исправлений для кода.

3.3.6. Распаковка кода

В процессе сборки задача do_unpack распаковывает исходные коды в каталог, указанный переменной ${S}. Если файлы исходных кодов загружались в виде архива, внутренняя структура которого соответствует соглашениям для каталога верхнего уровня ${BPN}-${PV}, установка S не требуется. Однако при указании в SRC_URI архива, не соблюдающего эти соглашения, или выборке файлов из SCM (скажем, Git или Subversion) задание должно указывать S. После извлечения файлов нужно убедиться в том, что каталог, указанный переменной ${S}, соответствует структуре исходных кодов.

3.3.7. Применение исправлений для кода

Иногда требуется после извлечения кода применить к нему исправления (patch). Все упомянутые в SRC_URI файлы .patch и .diff, а также архивы с такими файлами (например, .diff.gz) считаются исправлениями, которые задача do_patch применяет автоматически. Система сборки должна иметь возможность применять исправления с опцией -p1 (т. е. один уровень каталогов в пути удаляется). Если требуется удалять несколько уровней, их число указывается в опции striplevel для исправления в записи SRC_URI. Если исправление нужно применить в конкретном каталоге, не указанном в patch-файле, используется опция patchdir.

Как и всех локальные файлы, указанные в SRC_URI с помощью file://, patch-файлы следует размещать в каталоге с заданием, имя которого совпадает с базовым именем задания (BP и BPN), или в каталоге files.

3.3.8. Лицензирование

В задании должны быть определены переменные LICENSE и LIC_FILES_CHKSUM.

LICENSE

Указывает лицензию для программ. Если лицензия не известна, следует найти нужную информацию в исходном коде программ. Обычно лицензия указывается в файле COPYING, LICENSE или README. Зачастую лицензия также указывается в начале файлов с исходным кодом. Если программа распространяется по лицензии GNU General Public License version 2, следует установить LICENSE = "GPLv2".

Имена лицензий в файле LICENSE не должны включать пробелы, поскольку пробелы служат разделителями имён. Для стандартных лицензий используются имена файлов из каталога meta/files/common-licenses/ или имена флагов SPDXLICENSEMAP в файле meta/conf/licenses.conf.

LIC_FILES_CHKSUM

Эту переменную система сборки OE применяет для контроля неизменности текста лицензии. При наличии изменений процесс сборки завершится ошибкой.

Нужно указывать все файлы лицензий, применимых к программе. В конце этапа настройки конфигурации процесс сборки проверяет контрольные суммы файлов с текстами лицензий и выдаёт ошибку при несоответствии. Дополнительная информация о установке переменной LIC_FILES_CHKSUM приведена в параграфе 3.32.1. Отслеживание изменений в лицензиях.

Для определения корректных контрольных сумм файлов лицензий можно указать эти файлы в переменной LIC_FILES_CHKSUM с произвольными строками md5 и запустить сборку программ. В сообщениях об ошибках будут указаны корректные контрольные суммы md5. Дополнительная информация о контрольных суммах приведена в параграфе 3.3.5. Извлечение кода.

Ниже приведён пример переменной для программы, имеющей файл COPYING.

```
LIC_FILES_CHKSUM = "file://COPYING;md5=xxx"
```

При попытке собрать программу система сборки выдаст сообщение об ошибке, содержащее корректную контрольную сумму, которую нужно подставить в показанную в примере строку до следующей сборки.

3.3.9. Зависимости

Большинство программных пакетов имеют короткий список пакетов, которые нужны им. Эти пакеты называют зависимостями и они делятся на две категории - зависимости при сборке и зависимости при работе. В задании зависимости при сборке указываются с помощью переменной DEPENDS, куда следует включать имена других заданий (есть нюансы). Важно задать зависимости при сборке в явном виде. Если этого не сделать, в результате параллельной работы BitBake могут возникать конфликты, когда зависимость будет видна в одной задаче (например, do_configure) и отсутствовать в другой (например, do_compile).

Другое соображение связано с тем, что сценарии настройки могут автоматически проверять дополнительные зависимости и включать при их обнаружении соответствующую функциональность. Это означает, для для получения детерминированных результатов и отсутствия конфликтов нужно явно указать эти зависимости или явно запретить упомянутые добавления конфигурационным сценариям. Если нужно сделать задание более полезным (например, при публикации уровня) вместо жёсткого запрета функциональности можно использовать переменную PACKAGECONFIG, позволяющую легко включать или отключать функциональность и соответствующие зависимости другим людям.

Аналогичным способом указываются и зависимости при работе с помощью зависящих от пакета переменных RDEPENDS. Все зависящие от пакета переменные должны иметь имя пакета в качестве суффикса как переопределение. Поскольку имя основного пакета в задании совпадает с именем задания, которое можно узнать из переменной ${PN}, можно указывать зависимости для основного пакета в форме RDEPENDS_${PN}. Если пакет называется ${PN}-tools, переменная будет называться RDEPENDS_${PN}-tools и т. п.

Некоторые зависимости при работе задаются автоматически при упаковке. Такие зависимости включают все общие библиотеки (если пакет example включает libexample, а другой пакет mypackage - двоичные файлы, связанные с libexample, система сборки OE будет автоматически добавлять зависимость mypackage от example). Дополнительная информация о добавлении зависимостей приведена в разделе Automatically Added Runtime Dependencies [1].

3.3.10. Настройка конфигурации задания

Большинство программ включает те или иные средства настройки конфигурации перед сборкой. Обычно для этого применяется сценарий configure с определёнными опциями или изменяется конфигурационный файл сборки. Начиная с выпуска YP 1.7, в части основных заданий двоичные сценарии настройки конфигурации были отключены по причине многочисленных ошибок с подстановками путей. Сейчас система сборки OE использует более надёжный инструмент pkg-config. Список отключённых сценариев настройки приведён в разделе Binary Configuration Scripts Disabled [3].

Основной частью настройки конфигурации при сборке является проверка зависимостей и возможного включения свойств в результате этой проверки. Нужно задать все зависимости для сборки в переменной задания DEPENDS, указывая задания, требуемые зависимостями. Зависимости для сборки зачастую описаны в документации программ.

Ниже перечислены элементы настройки, применяемые в зависимости от способа сборки программ

Autotools

Если в исходных кодах имеется файл configure.ac, сборка происходит на основе autotools и для настройки нужно изменять конфигурацию. В этом случае задание должно наследовать класс autotools и не будет включать задачу do_configure. Тем не менее внесение корректив возможно. Например, можно задать переменную EXTRA_OECONF или PACKAGECONFIG_CONFARGS для передачи параметров конфигурации задания.

Cmake

Если в исходном коде имеется файл CMakeLists.txt file, сборка выполняется с использованием Cmake. В этом случае может потребоваться внесение изменений в конфигурацию. При использовании Cmake задания должны наследовать класс cmake и не будут включать задачу do_configure. Можно внести некоторые изменения с помощью переменной EXTRA_OECMAKE для передачи конфигурационных опций задания.

Прочее

Если исходные коды не включают файлов configure.ac и CMakeLists.txt, этого говорит о сборке программы с использованием методов, отличных от Autotools и CMake. В таких случаях обычно требуется включить задачу do_configure, если заданию нужны какие-либо настройки.

Если задание не использует Autotools или Cmake, настройка конфигурации может потребоваться и нужно понять, нужна ли она на самом деле. Может потребоваться редактирование файла Makefile или иного файла конфигурации, содержащего нужные для сборки опции. Иногда может потребоваться выполнение включённого в исходный код сценария настройки. В этом случае команда ./configure --help позволит увидеть доступные опции.

По завершении настройки конфигурации рекомендуется посмотреть файл log.do_configure, чтобы убедиться в установке нужных параметров и отсутствии дополнительных зависимостей при сборке, которые нужно добавить в DEPENDS. Например, если сценарий настройки нашёл что-то, не упомянутое в DEPENDS, или не нашёл нужного для обеспечения желаемой функциональности, нужно внести соответствующие добавления в DEPENDS. Просмотр журнала также может выявить элементы, которые были проверены и/или включены в конфигурацию, но не требуются, или не найдены DEPENDS, что может потребовать передачи дополнительных параметров сценарию настройки. Для просмотра полного списка параметров configure служит команда ./configure --help, вызванная из каталога ${S}.

3.3.11. Использование заголовочных файлов для интерфейса с устройствами

Если задание создаёт программу, которой нужно взаимодействовать с тем или иным устройством, или требуется API для пользовательского ядра, нужны соответствующие файлы заголовков. Ни при каких обстоятельствах недопустимо менять файл meta/recipes-kernel/linux-libc-headers/linux-libc-headers.inc, поскольку эти заголовочные файлы используются для сборки libc и не должны подвергаться рискам со стороны пользовательских или машинозависимых конфигураций. Настройка libc с изменёнными файлами заголовков будет влиять на все приложения, применяющие libc.

Корректным способом взаимодействия с устройством или пользовательским ядром является использование отдельного пакета, обеспечивающего дополнительные заголовки для драйвера или других уникальных интерфейсов. В этом случае данное приложение отвечает за создание зависимостей.

Предположим, что изменяется имеющийся заголовок, который добавляет управление вводом-выводом или поддержку сети. Если изменение используется небольшим числом программ, предоставление заголовку уникальной версии проще и оказывает меньшее влияние. А этом случае следует учитывать приведённые выше рекомендации.

Если по какой-то причине нужно изменить поведение libc и, следовательно, всех приложений в системе, нужно использовать файл .bbappend для изменения файла linux-kernel-headers.inc. Однако следует учесть, что изменения не должны быть машинозависимыми.

Рассмотрим вариант старого ядра, который требует применения старого libc ABI. Заголовки, установленные заданием, должны относиться к стандартному ядру, а не к пользовательскому. Для пользовательских заголовков ядра нужно получить их из переменной STAGING_KERNEL_DIR, указывающей каталог с заголовками, которые нужны для сборки внешних модулей. Задание должно включать do_configure[depends] += "virtual/kernel:do_shared_workdir".

3.3.12. Компиляция

В процессе сборки после извлечения, распаковки и настройки конфигурации исходного кода запускается задача do_compile. Если эта задача выполняется без ошибок, дополнительно ничего делать не нужно.

Однако при отказах в процессе компиляции приходится искать причины ошибок. При обнаружении неверных путей в конфигурационных файлах или невозможности найти библиотеки или файлы заголовков убедитесь в использовании надёжного пакета pkg-config (см. 3.3.10. Настройка конфигурации задания).

- *Отказы при параллельной сборке* проявляются как чередующиеся ошибки или сообщения об отсутствии файлов или каталогов, которые должны быть созданы другой частью процесса сборки. Это может быть связано с некорректным порядком выполнения процесса сборки. Для решения проблемы следует обеспечить невыполненные зависимости в Makefile, создавшем Makefile сценарии или (обходной путь) отключить параллельную сборку, указав PARALLEL_MAKE = "" (см. 3.30.12. Отладка конфликтов параллельной сборки).
- *Некорректное использование путей хоста* может возникать при сборке заданий для целевой платформы или nativesdk. Ошибка возникает при использовании процессом компиляции некорректных заголовков, библиотек или иных файлов хост-системы при кросс-компиляции для целевой платформы.

  Для решения проблемы следует просмотреть файл log.do_compile, где нужно найти используемые пути хоста (/usr/include, /usr/lib и т. п.) и исправить соответствующие опции конфигурации или применить исправления.
- *Отказ при поиске нужных библиотек и заголовочных файлов.* Если зависимости при сборке не выполняются по причине их отсутствия в DEPENDS или использования неверных путей поиска, процесс сборки завершается ошибкой. Система сборки в таких случаях указывает не найденные файлы. Для решения проблему нужно указать пропущенные зависимости или дополнительные параметры сценариев настройки конфигурации.

  Иногда нужно внести изменения в исходные коды для обеспечения корректных путей поиска. Если нужно указать пути поиска файлов из других заданий, размещённых в sysroot, следует использовать переменные системы сборки OE (например, STAGING_BINDIR, STAGING_INCDIR, STAGING_DATADIR и т. п.).

3.3.13. Инсталляция

Задача do_install в процессе работы копирует файлы сборки вместе с иерархией в места, отражающие их размещение на целевой платформе. Файлы копируются из каталогов ${S}, ${B} и ${WORKDIR} в каталог ${D} для создания структуры, которая должна появиться в целевой системе.

Сборка программ влияет на операции, требуемые для корректной установки. Ниже описаны действия, зависящие от способа сборки программ.

- *Autotools и Cmake.* Если программы вашего задания собираются с помощью Autotools или CMake, система сборки OE знает, как их установить. Поэтому не требуется включать задачу do_install в задание, достаточно

лишь убедиться в полноте сборки и отсутствии ошибок. Однако, если нужно установить дополнительные файлы, которые не учтены командой make install, следует использовать функцию do_install_append с командой install, как описано ниже.

- *Прочие (make install).* В задании нужно определить функцию do_install, которая должна вызывать oe_runmake install и вероятно будет передавать ей целевой каталог. Способ передачи каталога зависит от файла Makefile (например, DESTDIR=${D}, PREFIX=${D}, INSTALLROOT=${D} и т. п.). Пример задания, использующего make install приведён в параграфе 3.3.21.3. Пакеты на основе Makefile.
- *Ручная установка.* В задании нужно определить функцию do_install, которая должна использовать команду install -d для создания каталогов в ${D}. Когда каталоги созданы, функция может использовать команду install для установки программ в эти каталоги.

  Дополнительную информацию можно найти по ссылке.

Для сценариев, не использующих Autotools или CMake, нужно отслеживать инсталляцию, контролируя и исправляя возникающие ошибки. Нужно просматривать каталог ${D} (по умолчанию ${WORKDIR}/image) для проверки корректности установки всех файлов.

- В процессе инсталляции может потребоваться изменение некоторых установленных файлов в соответствии с целевой платформой. Например, может потребоваться замена жёстко заданных путей значениями подставляемых системой переменных (например, замена /usr/bin/ на ${bindir}). Если такие замены происходили в процессе do_install, нужно менять целевой файл после копирования, а не до него. Это гарантирует возможность повтора системой сборки задачи do_install при возникновении необходимости.
- Задача oe_runmake, которая может запускаться напрямую или опосредованно через класса autotools и cmake, запускает команды make install параллельно. Иногда в файле Makefile могут отсутствовать зависимости между целями и будут возникать конфликты. При возникновении сбоев в процессе выполнения do_install можно отключить параллельную работу, задав PARALLEL_MAKEINST = "".

3.3.14. Включение системных служб

Для включения службы, которая будет запускаться при загрузке и работать в фоновом режиме, нужно указать в задании дополнительные определения. Если добавляется служба, а сценарий её запуска или файл самого сервиса не установлен, нужно обеспечить соответствующую инсталляцию с использованием задачи do_install_append. Если задание уже включает эту функцию, следует добавить данные в конец строки, а не добавлять другую функцию.

При настройке инсталляции служб нужно выполнить операции, которые обычно делает команда make install. Иными словами, нужно убедиться, что при инсталляции данные упорядочиваются как в целевой системе.

Система сборки OE обеспечивает несколько вариантов запуска служб при загрузке.

- *SysVinit* является менеджером системы и служб, который управляет инициализацией системой init для контроля базовых функций. Программа init является первой программой, запускаемой ядром Linux при загрузке системы. Затем эта программа управляет запуском, работой и отключением всех прочих программ.

  Для включения службы с помощью SysVinit задание должно наследовать класс update-rc.d, который помогает безопасно устанавливать пакеты на целевой платформе. В задании нужно установить переменные INITSCRIPT_PACKAGES, INITSCRIPT_NAME и INITSCRIPT_PARAMS.
- *systemd* является демоном управления системой, предназначенным для замены SysVinit и обеспечения более эффективного управления службами (см. http://freedesktop.org/wiki/Software/systemd/). Для управления службами с помощью systemd задание должно наследовать класс systemd, информацию о котором можно найти в файле systemd.bbclass дерева исходных кодов.

3.3.15. Подготовка пакетов

Успешная подготовка пакетов включает автоматизированные процессы системы сборки OE и действия пользователя.

- *Разделение файлов.* Задача do_package делит файлы, созданные заданием на логические компоненты. Даже программа из одного двоичного файла может включать отладочные символы, документацию и другие компоненты, которые следует разделять.
- *Запуск тестов QA.* Класс insane добавляет в процесс генерации пакета этап создания тестов качества системой сборки OE. На этом этапе выполняется множество проверок, чтобы гарантировать отсутствие в выводе типичных проблем, возникающих при работе. Информация об этих проверках приведена в описании класса insane и разделе QA Error and Warning Messages [3].
- *Проверка пакетов вручную.* После сборки программы нужно убедиться в корректности пакетов. Проверяется каталог ${WORKDIR}/packages-split на предмет наличия ожидаемых файлов. При обнаружении проблем следует проверить корректность установки PACKAGES, FILES, do_install(_append) и т. п..
- *Разделение приложения на несколько пакетов* (см. 3.3.21.4. Разделение программы на несколько пакетов).
- *инсталляция сценария пост-установки* (см. 3.3.19. Сценарии пост-установки).
- *Указание архитектуры пакета.* В зависимости от того, что собирает задание и как оно настроено может быть важна маркировка пакетов как относящихся к конкретной машине или архитектуре. По умолчанию пакеты применимы к любой машине с архитектурой, указанной для цели. Когда задание создаёт машинозависимые пакеты (например, значение MACHINE передаётся сценарию configure или применяется patch-файл для определённой машины), нужно помечать пакеты, указывая PACKAGE_ARCH = "${MACHINE_ARCH}" в задании.

  Если задание создаёт пакеты, которые не привязаны к определённой машине или архитектуре (например, файлы сценариев или конфигурации), следует использовать класс allarch, добавляя в задание inherit allarch.

Проверка корректности архитектуры пакета не является критичной для первых сборок задания. Однако она важна для обеспечения гарантий надлежащей перестройки задания при смене конфигурации. Особенно важно контролировать установку на платформе подходящих пакетов в случаях сборки для нескольких целей.

3.3.16. Совместное использование файлов заданиями

Заданиям зачастую нужны файлы из других заданий сборочного хоста. Например, приложение может быть связано с общей библиотекой и ему нужен доступ к самой библиотеке и файлам заголовков. Этот доступ осуществляется через файловую систему sysroot. Каждое задание имеет в рабочем каталоге две системы sysroot, одна из которых служит для целевых файлов (recipe-sysroot), другая для естественных файлов хоста сборки (recipe-sysroot-native).

Заданиям не следует заполнять sysroot напрямую (т. е. записывать туда файлы). Вместо этого файлы должны устанавливаться в стандартные места задачей do_install внутри каталога ${D}. Причина этого заключается в том, что практически все файлы sysroot указываются в манифестах для контроля последующего изменения или удаления заданий. Поэтому система sysroot способна избавляться от устаревших файлов.

Часть файлов, установленных задачей do_install, используется задачей do_populate_sysroot в соответствии с переменной SYSROOT_DIRS для автоматического заполнения sysroot. Список каталогов sysroot можно менять. Например, можно добавить каталог /opt в форме SYSROOT_DIRS += "/opt". Более подробное описание задачи do_populate_sysroot и связанных с ней функций приведено в описании класса staging.

3.3.17. Использование виртуальных провайдеров

Если до сборки известно, что одну функциональность представляет несколько разных заданий, можно выбрать виртуального провайдера (virtual/*) в качестве замены реального, который будет определён в процессе сборки. Базовым вариантом применения виртуальных провайдеров являются задания для сборки ядра. Предположим наличие трёх таких заданий, у которых значения PN отображаются на kernel-big, kernel-mid и kernel-small. Кроме того, каждое из этих заданий своим способом использует оператор PROVIDES, указывая свою возможность предоставить элемент virtual/kernel. Одним из способов прохождения класса kernel является

```
PROVIDES += "${@ "virtual/kernel" if (d.getVar("KERNEL_PACKAGE_NAME") == "kernel") else "" }"
```

Любое задание, наследующее класс kernel, использует оператор PROVIDES, который указывает, что задание может обеспечить элемент virtual/kernel.

Когда нужно собрать образ и выбрать для него ядро, можно настроить сборку с нужным ядром с помощью переменной PREFERRED_PROVIDER. Рассмотрим в качестве примера файл x86-base.inc, задающий конфигурацию машины (MACHINE). Этот включаемый файл задаёт для всех машин x86 использование ядра linux-yocto. Это задаётся приведёнными ниже строками.

```
PREFERRED_PROVIDER_virtual/kernel ??= "linux-yocto"
PREFERRED_VERSION_linux-yocto ??= "4.15%"
```

При использовании виртуального провайдера не нужно жёстко задавать имя задания в качестве зависимости при сборке. Можно использовать переменную DEPENDS для указания зависимости от virtual/kernel, например, DEPENDS = "virtual/kernel".

В процессе сборки система OE выберет нужное задание для зависимости virtual/kernel на основе переменной PREFERRED_PROVIDER. Для использования упомянутого выше компактного ядра можно указать PREFERRED_PROVIDER_virtual/kernel ??= "kernel-small".

Любое задание, где PROVIDES содержит элемент virtual/*, который не выбран с помощью PREFERRED_PROVIDER, собираться не будет. Предотвращение сборки таких заданий обычно желательно, поскольку цель этого механизма заключается в выборе одного из взаимоисключающих провайдеров. Примеры виртуальных провайдеров даны ниже.

- *virtual/kernel* обеспечивает имя задания для ядра, используемого при сборке образа ядра.
- *virtual/bootloader* обеспечивает имя загрузчика (bootloader), используемого при сборке образа ядра.
- *virtual/mesa* обеспечивает gbm.pc.
- *virtual/egl* обеспечивает egl.pc и возможно wayland-egl.pc.
- *virtual/libgl* обеспечивает gl.pc (libGL).
- *virtual/libgles1* обеспечивает glesv1_cm.pc (libGLESv1_CM).
- *virtual/libgles2* обеспечивает glesv2.pc (libGLESv2).

3.3.18. Корректные версии предварительных выпусков

Иногда имя задания может приводить к проблемам с версиями при обновлении до финального выпуска задания. Рассмотрим, например, файл irssi_0.8.16-rc1.bb recipe из примеров параграфа 3.3.3. Сохранение и именование заданий. Это задание указывает предварительный выпуск (rc1), а в окончательном выпуске файл получает имя irssi_0.8.16.bb. Версия 0.8.16-rc1 сменилась на 0.8.16, что представляется уменьшением номера с точки зрения менеджера пакетов, поэтому полученные в результате пакеты не будут корректно обновляться.

Для корректного сравнения версий рекомендуется для PV в задании установить previous_version+current_version. Можно использовать дополнительную переменную, чтобы текущая версия была доступна везде. Например,

```
REALPV = "0.8.16-rc1"
PV = "0.8.15+${REALPV}"
```

3.3.19. Сценарии пост-установки

Сценарии пост-установки работают сразу после инсталляции пакета на целевой платформе или при создании образа, когда пакет включён в образ. Для включения в пакет такого сценария нужно добавить функцию pkg_postinst_PACKAGENAME() в файл задания (.bb), заменив PACKAGENAME именем соответствующего пакета. Для

применения сценария пост-установки к основному пакету задания (обычно это нужно), вместо PACKAGENAME указывается ${PN}.

Структура функции пост-установки показана ниже.

```
pkg_postinst_PACKAGENAME() {
# Выполняемые команды
}
```

Сценарий, заданный в функции пост-установки, вызывается при создании корневой файловой системы. При успешном выполнении сценария пакет помечается как установленный. Если возникает отказ сценария, пакет помечается как нераспакованный и сценарий повторяется при следующей загрузке.

Всем пост-установочным сценариям RPM на целевой платформе следует возвращать код завершения 0. RPM не разрешает отличные от 0 коды возврата и менеджер пакетов RPM в таких случаях выдаёт отказ при установке на целевой платформе.

Иногда нужно отложить сценарий пост-установки до первой перезагрузки (например, сценарию может требоваться выполнение на самом устройстве). В таких случаях нужно явно указать отложенные до перезагрузки сценарии, для чего можно использовать функцию pkg_postinst_ontarget() или вызвать postinst-intercepts defer_to_first_boot из pkg_postinst(). Любой отказ сценария pkg_postinst() (включая exit 1) вызывает ошибку задачи do_rootfs.

При наличии заданий, использующих функцию pkg_postinst и требующих нестандартных естественных инструментов, которые имеют зависимости при создании корневой файловой системы, нужно указывать эти инструменты в переменной задания PACKAGE_WRITE_DEPS. Если переменная не используется, задание может отсутствовать и выполнение пост-установочного сценария будет отложено до перезагрузки, что нежелательно, а на файловых системах с доступом только для чтения - невозможно.

Существует эквивалентная поддержка сценариев pre-install, pre-uninstall и post-uninstall с помощью функций pkg_preinst, pkg_prerm и pkg_postrm, соответственно. Эти сценарии работают так же, как pkg_postinst за исключением того, что они запускаются в разные моменты. Кроме того, с учётом момента запуска эти сценарии не применимы во время создания образа, как pkg_postinst.

3.3.20. Тестирование

Финальным этапом задания является проверка корректности работы сборки. Для тестирования добавьте выходные пакеты в образ и проверьте их на целевой платформе. Информация о добавлении пакетов в образы приведена в разделе 3.2. Настройка образов.

3.3.21. Примеры

Для иллюстрирования процесса написания заданий в этом разделе приведено несколько примеров:

- задания, использующие локальные файлы;
- задания, использующие автоматически настраиваемые пакеты;
- задания, использующие пакеты с Makefile;
- расщепление приложения на несколько пакетов;
- добавление двоичных файлов в образ.

3.3.21.1. Пакет с одним файлом .c

Сборка приложения из одного локального файл требует задания, где этот файл указан в переменной SRC_URI. Кроме того, нужно вручную создать задачи do_compile и do_install. Переменная S определяет каталог с исходным кодом, который задаётся в качестве WORKDIR в данном случае. BitBake использует этот каталог для сборки.

```
SUMMARY = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
   ${CC} helloworld.c -o helloworld
}

do_install() {
   install -d ${D}${bindir}
   install -m 0755 helloworld ${D}${bindir}
}
```

По умолчанию собираются пакеты helloworld, helloworld-dbg и helloworld-dev. Информация о настройке процесса упаковки приведена в параграфе 3.3.21.4. Разделение программы на несколько пакетов.

3.3.21.2. Пакеты с автоматической настройкой

Приложения, использующие инструменты автоматической настройки, такие как autoconf и automake, требуют от заданий указать архив исходных кодов в переменной SRC_URI и наследовать класс autotools, который содержит определения всех этапов сборки приложений с автоматической настройкой. Результат сборки автоматически собирается в пакет. Если приложение использует NLS для поддержки разных языков, создаются языковые пакеты (по одному на язык). Ниже приведён пример на основе задания hello_2.3.bb.

```
SUMMARY = "GNU Helloworld application"
```

```
SECTION = "examples"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=751419260aa954499f7abaabaa882bbe"

SRC_URI = "${GNU_MIRROR}/hello/hello-${PV}.tar.gz"

inherit autotools gettext
```

Переменная LIC_FILES_CHKSUM служит для отслеживания изменений в лицензиях, как описано в разделе Tracking License Changes [1]. Задания для автоматически настраиваемых приложений можно создавать по аналогии с предыдущим параграфом.

3.3.21.3. Пакеты на основе Makefile

Приложения, использующие GNU, также требуют от заданий указать архив исходных кодов в переменной SRC_URI. Этап do_compile добавлять не нужно, поскольку BitBake по умолчанию запускает команду make для компиляции приложения. Если нужны дополнительные опции make, следует задать их в переменной EXTRA_OEMAKE или PACKAGECONFIG_CONFARGS. BitBake передаёт эти опции при вызове GNU make. Отметим, что задача do_install по-прежнему требуется, иначе BitBake будет запускать пустую задачу do_install.

Некоторые приложения могут требовать передачи компилятору дополнительных параметров (например, путь к дополнительным файлам заголовков). Это можно обеспечить, добавив параметры в переменную CFLAGS в форме CFLAGS_prepend = "-I ${S}/include ".

Ниже приведён пример приложения mtd-utils использующего Makefile.

```
SUMMARY = "Tools for managing memory technology devices"
SECTION = "base"
DEPENDS = "zlib lzo e2fsprogs util-linux"
HOMEPAGE = "http://www.linux-mtd.infradead.org/"
LICENSE = "GPLv2+"
LIC_FILES_CHKSUM = "file://COPYING;md5=0636e73ff0215e8d672dc4c32c317bb3 \
file://include/common.h;beginline=1;endline=17;md5=ba05b07912a44ea2bf81ce409380049c"

# Use the latest version at 26 Oct, 2013
SRCREV = "9f107132a6a073cce37434ca9cda6917dd8d866b"
SRC_URI = "git://git.infradead.org/mtd-utils.git \
    file://add-exclusion-to-mkfs-jffs2-git-2.patch \
"

PV = "1.5.1+git${SRCPV}"

S = "${WORKDIR}/git"

EXTRA_OEMAKE = "'CC=${CC}' 'RANLIB=${RANLIB}' 'AR=${AR}' 'CFLAGS=${CFLAGS} -I${S}/include -DWITHOUT_XATTR' 'BUILDDIR=${S}'"

do_install () {
 oe_runmake install DESTDIR=${D} SBINDIR=${sbindir} MANDIR=${mandir} INCLUDEDIR=${includedir}
}

PACKAGES =+ "mtd-utils-jffs2 mtd-utils-ubifs mtd-utils-misc"

FILES_mtd-utils-jffs2 = "${sbindir}/mkfs.jffs2 ${sbindir}/jffs2dump ${sbindir}/jffs2reader ${sbindir}/sumtool"
FILES_mtd-utils-ubifs = "${sbindir}/mkfs.ubifs ${sbindir}/ubi*"
FILES_mtd-utils-misc = "${sbindir}/nftl* ${sbindir}/ftl* ${sbindir}/rfd* ${sbindir}/doc* ${sbindir}/serve_image ${sbindir}/recv_image"

PARALLEL_MAKE = ""

BBCLASSEXTEND = "native"
```

3.3.21.4. Разделение программы на несколько пакетов

Чтобы разделить приложение на несколько пакетов, можно использовать переменные PACKAGES и FILES. Ниже приведён пример разделения задания libxpm, которое по умолчанию создаёт один пакет с библиотекой и несколькими исполняемыми файлами. Можно разделить задание, выделив исполняемые файлы в отдельные пакеты.

```
require xorg-lib-common.inc
SUMMARY = "Xpm: X Pixmap extension library"
LICENSE = "BSD"
LIC_FILES_CHKSUM = "file://COPYING;md5=51f4270b012ecd4ab1a164f5f4ed6cf7"
DEPENDS += "libxext libsm libxt"
PE = "1"

XORG_PN = "libXpm"

PACKAGES =+ "sxpm cxpm"
FILES_cxpm = "${bindir}/cxpm"
FILES_sxpm = "${bindir}/sxpm"
```

В этом примере исполняемые файлы sxpm и cxpm вынесены в отдельные пакеты. Поскольку bindir помещается по умолчанию в основной пакет PN, дополнительные пакеты помещены в начало переменной PACKAGES. Это создаёт дополнительные переменные FILES_*, содержащие информацию о файлах и каталогах, включаемых в пакеты. Включённые предыдущими пакетами файлы и каталоги пропускаются последующими, поэтому указанные в переменных файлы не попадут в основной пакет PN.

3.3.21.5. Упаковка внешних двоичных пакетов

Иногда нужно добавить в образ собранные заранее двоичные файлы. Например, при наличии фирменного (proprietary) кода, созданного отдельным подразделением компании, эти программы могут оказаться нужны для образа, создаваемого в системе сборки OE. Поскольку имеются лишь двоичные файлы без исходных кодов, невозможно использовать типовые задания для сборки, которым нужно указать исходный код в SRC_URI и компилировать его.

Одним из вариантов является упаковка двоичных файлов и установка их в образ. В общем случае это не лучшее решение, поскольку может мешать воспроизведению сборки и создать в будущем проблемы совместимости с ABI. Однако иногда других вариантов просто нет. Простейшим способом является подготовка задания, использующего класс bin_package с указанием принятого по умолчанию размещения результатов сборки. В большинстве случаев класс bin_package «пропускает» этапы настройки и компиляции, а также настраивает извлечение пакетов из указанных мест. В частности, этот класс устанавливает noexec для задач do_configure и do_compile, задаёт FILES_${PN} = "/" для выборки файлов и настраивает задачу do_install для копирования файлов из ${S} в ${D}. Класс bin_package хорошо работает, когда извлечённые в ${S} файлы уже размещены в соответствии с их расположением на целевой платформе. Информация о переменных FILES, PN, S и D приведена в [3].

- Использование DEPENDS является хорошей идеей для компонент, распространяемых в двоичной форме, и зачастую требуется для общих библиотек. Для библиотек указание зависимостей в DEPENDS обеспечивает доступность этих библиотек при подготовке sysroot, когда другие задания ссылаются на эти библиотеки.
- Использование DEPENDS также позволяет задать автоматическое добавление зависимостей при работе программ (см. раздел Automatically Added Runtime Dependencies [1]).

Если нет возможности использовать класс bin_package, нужно обеспечить выполнение приведённых ниже операций.

- Подготовить задание, где do_configure и do_compile не делают ничего. Обычно для этого достаточно просто не указывать задачи, поскольку принятые по умолчанию реализации нечего не делают, пока не найден файл Makefile в ${S}.

  Если каталог ${S} может включать файл Makefile или нужно наследовать класс, который заменит do_configure и do_compile настроенными вариантами, можно использовать флаг [noexec] для включения задач в no-ops.

  ```
  do_configure[noexec] = "1"
  do_compile[noexec] = "1"
  ```

  В отличие от удаления задач, использование флага сохранит цепочку зависимостей из задач do_fetch, do_unpack и do_patch для задачи do_install.
- Убедиться в корректной установке двоичных файлов задачей do_install.
- Убедиться, что переменная FILES (обычно FILES_${PN}) указывает устанавливаемые файлы. Это зависит от места установки файлов с учётом принятого по умолчанию размещения.

3.3.22. Рекомендации по стилям заданий

При подготовке заданий разумно следовать сложившемуся стилю. На странице OpenEmbedded Styleguide даны рекомендации по предпочтительному стилю заданий. Реальные задания обычно слегка отклоняются от рекомендуемого стиля, однако стремление следовать стилю является хорошим тоном. Отсутствие пробелов вокруг операторов = при назначении или некорректный порядок компонент задания часто воспринимаются как дурной тон.

3.3.23. Синтаксис заданий

Для подготовки задания важно разобраться с их синтаксисом. Ниже приведён обзор основных элементов файла заданий BitBake, более полная документация содержится в разделе Syntax and Operators [6].

- *Назначение и изменение переменных.* Назначение может быть статическим текстом или включать содержимое других переменных. Кроме того, поддерживаются операторы добавления в начало или в конец переменной.

  ```
  S = "${WORKDIR}/postfix-${PV}"
  CFLAGS += "-DNO_ASM"
  SRC_URI_append = " file://fixup.patch"
  ```
- *Функции* задают последовательность выполняемых операций. Обычно функции применяются для переопределения принятой по умолчанию реализации функции задачи или её дополнения (т. е. добавления в начало или конец имеющейся функции). Стандартные функции используют стиль командного процессора sh, но поддерживается также доступ к переменным и внутренним методам OE. Ниже приведён пример функции из задания sed.

  ```
  do_install () {
      autotools_do_install
      install -d ${D}${base_bindir}
      mv ${D}${bindir}/sed ${D}${base_bindir}/sed
      rmdir ${D}${bindir}/
  }
  ```

  Можно реализовать новые функции, которые вызываются между имеющимися задачами, если эти функции не заменяют и не дополняют принятых по умолчанию функций. Можно реализовать функции на языке Python.
- *Ключевые слова.* Задания BitBake используют немного ключевых слов. Они служат для включения базовых функций (inherit), загрузки частей задания из других файлов (include и require), а также для экспорта переменных в среду (export).

  ```
  export POSTCONF = "${STAGING_BINDIR}/postconf"
  inherit autoconf
  require otherfile.inc
  ```
- *Комментарии (#).* Все строки, начинающиеся с символа #, считаются комментариями и не обрабатываются.

  ```
  # Это комментарий
  ```

Далее представлены наиболее важные и распространённый части синтаксиса заданий. Дополнительная информация о синтаксисе приведена в разделе Syntax and Operators [6].

- *Продолжение строки (\).* Символ \ служит для разделения длинной строки на несколько более коротких.

  ```
  VAR = "Это длинная \
  строка"
  ```

Этот символ нельзя заменить другим.

- *Использование переменных (${VARNAME}).* Синтаксис ${VARNAME} служит для доступа к содержимому переменных, как показано ниже

```
SRC_URI = "${SOURCEFORGE_MIRROR}/libpng/zlib-${PV}.tar.gz"
```

Важно понимать, что значение переменной, выраженное в такой форме, не подставляется автоматически сразу. Преобразование выражений происходит позже по запросу (обычно при выполнении ссылающейся на переменную функции). Это обеспечивает гарантию того, что значения больше подходят для контекста, в котором они реально применяются. В редких случаях, когда требуется сразу раскрыть выражение, можно использовать оператор := вместо = при назначении.

- *Назначения в кавычках ("value").* Присваиваемые значения следует указывать в двойных кавычках, например,

```
VAR1 = "${OTHERVAR}"
VAR2 = "Версия ${PV}"
```

- *Условное назначение (?=)* используется для присваивания переменной значения исключительно в случае его отсутствия в данный момент. Для условного (мягкого) назначения используется оператор ?=). Обычно такие назначения применяются в файле local.conf для переменных, которым разрешено приходить извне.

  Например, VAR1 ?= "New value" назначит переменной VAR1 значение "New value", если в этот момент она не имеет значения. Иначе переменная VAR1 сохранит своё значение как в приведённом ниже примере.

```
VAR1 = "Original value"
VAR1 ?= "New value"
```

- *Добавление в конце (+=).* Оператор += добавляет значение в конце имеющейся переменной, отделяя новое значение от имеющегося пробелом. Например, SRC_URI += "file://fix-makefile.patch".

- *Добавление в начале (=+).* Оператор =+ позволяет добавить значение в начало имеющейся переменной с пробелом между добавленным и прежним значением. Например, VAR =+ "Starts".

- *Добавление в конце (_append).* Оператор _append добавляет значение к имеющейся переменной без включения символа пробела. Оператор применяется после всех операторов += и =+, а также после назначений с помощью =. Чтобы добавленное значение не сливалось с имеющимся, следует явно указывать пробел в начале добавляемого значения, например, SRC_URI_append = " file://fix-makefile.patch".

  Оператор _append можно использовать в переопределениях, чтобы изменение относилось лишь к указанной цели или машине. Например, SRC_URI_append_sh4 = " file://fix-makefile.patch".

- *Добавление в начале (_prepend).* Оператор _prepend добавляет значение к имеющейся переменной без включения символа пробела. Оператор применяется после всех операторов += и =+, а также после назначений с помощью =. Чтобы добавленное значение не сливалось с имеющимся, следует явно указывать пробел в начале добавляемого значения, например, CFLAGS_prepend = "-I${S}/myincludes ".

  Оператор _prepend можно использовать в переопределениях, чтобы изменение относилось лишь к указанной цели или машине. Например, CFLAGS_prepend_sh4 = "-I${S}/myincludes ".

- *Переопределение* может служить для условной установки значения, обычно в зависимости от способа сборки задания. Например, для установки в переменной KBRANCH значения standard/base для любого целевого значения MACHINE, кроме qemuarm, где следует установить standard/arm-versatile-926ejs, можно задать

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

Переопределения применяются также для разделения вариантов переменной. Например, при установке таких переменных как FILES и RDEPENDS, которая зависит от собираемых заданием пакетов, всегда следует применять переопределение, задающее имя пакета.

- *Вставка пробелов в начале строки.* Для отступа в начале строки следует использовать пробелы, а не символы табуляции. Для функций оболочки ((shell) в настоящее время работают оба варианта, однако это просто правило YP по использованию табуляции в shell-функциях. В некоторых слоях (уровнях) может применяться иная трактовка отступов.

- *Комплексные операции Python.* Для более сложной обработки (например, поиск и замена в переменной) можно использовать при назначении переменных код Python, указываемого с помощью синтаксиса ${@python_code}, например, SRC_URI = "ftp://ftp.info-zip.org/pub/infozip/src/zip${@d.getVar('PV',1).replace('.', '')}.tgz.

- *Синтаксис shell-функций.* Эти функции создаются как сценарии командного процессора для описания последовательности выполняемых операций. Следует проверить работу сценария с базовым командным процессором sh и отсутствие потребности в использовании bash или иного процессора. Это относится и к применяемым системным утилитам (например, sed, grep, awk). При наличии сомнений следует проверить функцию с несколькими реализациями, включая BusyBox.

3.4. Добавление машины

Добавление новой машины в YP достаточно просто. В этом разделе описано, как добавить машины по аналогии с имеющимися в YP. Добавление полностью новой архитектуры может потребовать внесения изменений в gcc/glibc и информацию о сайте, но это выходит за рамки данного руководства. Полное описание процессов добавления новой машины приведено в разделе Creating a New BSP Layer Using the bitbake-layers Script [5].

3.4.1. Добавление конфигурационного файла машины

Для добавления машины нужно создать конфигурационный файл машины в каталоге conf/machine нужного уровня. Этот файл будет включать данные о добавляемом устройстве. Система сборки OE использует корневое имя конфигурационного файла машины для ссылок на неё. Например, при конфигурационном файле crownbay.conf система сборки будет называть машину crownbay.

Наиболее важными переменными в конфигурационном файле машины или включаемых в него файлах являются:

- TARGET_ARCH (например, "arm");
- PREFERRED_PROVIDER_virtual/kernel;
- MACHINE_FEATURES (например, "apm screen wifi").

Могут также потребоваться переменные:

- SERIAL_CONSOLES (например, "115200;ttyS0 115200;ttyS1");
- KERNEL_IMAGETYPE (например, "zImage");
- IMAGE_FSTYPES (например, "tar.gz jffs2").

В качестве образца можно взять один из файлов .conf каталога meta-yocto-bsp/conf/machine/.

3.4.2. Добавление ядра для машины

Система сборки OE должна быть способна собрать ядро для машины, что требует создать специальное задания для ядра или расширить имеющееся. Примеры заданий для ядер можно найти в каталоге исходных кодов (meta/recipes-kernel/linux) и использовать в качестве образца.

При подготовке нового задания для ядра применяются обычные правила установки SRC_URI. Таким образом, нужно указать все требуемые исправления (patch) и установить в S местоположение исходного кода. Нужно создать задачу do_configure для настройки распакованного ядра с использованием файла defconfig. Это можно сделать с помощью команды make defconfig или путём копирования подходящего файла defconfig и запуска команды make oldconfig. При использовании наследуемого ядра и, возможно, некоторых файлов linux-*.inc большая часть остальных функций будет централизована и принятые по умолчанию настройки классов обычно будут работать хорошо.

При расширении имеющегося задания обычно следует добавить подходящий файл defconfig. Файл нужно добавлять в каталог, похожий на каталоги размещения файлов defconfig других машин в данном задании для ядра. Это можно сделать путём включения файла с SRC_URI и добавления машины в выражение COMPATIBLE_MACHINE, например, COMPATIBLE_MACHINE = '(qemux86|qemumips)'.

Дополнительная информация о файлах defconfig приведена в разделе Changing the Configuration [7].

3.4.3. Добавление файла аппаратной конфигурации

Файл аппаратной конфигурации обеспечивает информацию о целевой платформе, для которой будет собираться образ, а также сведения, которые система сборки не может получить из других источников, таких как ядро. Примерами данных, включаемых в этот файл, служат ориентация буфера кадров, наличие клавиатуры, её расположение относительно экрана, а также экранное разрешение.

Система сборки в большинстве случаев подразумевает разумные значения, однако может потребоваться создание файла machconfig в каталоге meta/recipes-bsp/formfactor/files, содержащем данные для конкретных машин, таких как qemuarm и qemux86. Информацию о доступных и заданных по умолчанию значениях можно найти в файле meta/recipes-bsp/formfactor/files/config указанного каталога. Ниже приведён пример для машины qemuarm.

```
HAVE_TOUCHSCREEN=1
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
#DISPLAY_WIDTH_PIXELS=640
#DISPLAY_HEIGHT_PIXELS=480
#DISPLAY_BPP=16
DISPLAY_DPI=150
DISPLAY_SUBPIXEL_ORDER=vrgb
```

3.5. Обновление заданий

С течением времени разработчики публикуют новые версии программ, собираемых заданиями для уровней. Рекомендуется поддерживать актуальность кода таких заданий. Есть несколько способов обновления заданий, но сначала разумно проверить текущее состояние. Сделать это можно с помощью команды devtool check-upgrade-status (см. раздел Checking on the Upgrade Status of a Recipe [3]).

Далее рассмотрены три варианта обновления заданий - автоматическое, полуавтоматическое и ручное.

3.5.1. Использование AUH

Утилита AUH[1] работает с системой сборки OE для автоматической генерации обновлений на основе новых опубликованных версий кода. AUH позволяет создать службу автоматического обновления и может уведомлять о результатах по электронной почте. AUH может обновлять несколько заданий в один приём. Можно также протестировать результаты сборки и интеграции, используя образы, сохранённые на диске, с возможностью отправки результатов теста сопровождающим по электронной почте. Кроме того, AUH создаёт фиксации Git (commit) с сообщениями в дереве уровня для просмотра внесённых изменений.

В некоторых случаях не следует применять AUH для обновления, используя взамен команду devtool upgrade или обновление вручную.

- *AUH не может завершить последовательность обновлений*, что обычно является результатом невозможности автоматического переноса пользовательских исправлений в новую версию. Проблему решает обновление вручную.

[1]Auto Upgrade Helper - помощник для автоматического обновления.

- *Требуется более полный контроль процесса обновления*, например, при наличии специальных тестов.

Ниже перечислены этапы автоматического обновления с помощью AUH.

1. *Настройка хоста разработки* для работы с YP в соответствии с разделом 2.2. Подготовка сборочного хоста.
2. *Настройка Git.* Утилита AUH требует настройки конфигурации Git, поскольку этот пакет служит для сохранения изменений. Это требует указать пользователя Git и электронную почту. Для просмотра конфигурации служит команда git config —list. Если имя пользователя и электронная почта ещё не заданы, следует ввести команды

```
$ git config --global user.name some_name
$ git config --global user.email username@domain.com
```

3. *Клонирование репозитория AUH.* Для использования AUH нужно клонировать репозиторий на хост разработки, как показано ниже.

```
$ git clone git://git.yoctoproject.org/auto-upgrade-helper
Cloning into 'auto-upgrade-helper'...
remote: Counting objects: 768, done.
remote: Compressing objects: 100% (300/300), done.
remote: Total 768 (delta 499), reused 703 (delta 434)
Receiving objects: 100% (768/768), 191.47 KiB | 98.00 KiB/s, done.
Resolving deltas: 100% (499/499), done.
Checking connectivity... done.
```

AUH сейчас является частью репозиториев OpenEmbedded-Core (OE-Core) и Poky.

4. *Создание выделенного каталога сборки.* Сценарий oe-init-build-env создаёт свежий каталог сборки, который используется лишь для запуска утилиты AUH.

```
$ cd ~/poky
$ source oe-init-build-env your_AUH_build_directory
```

Использование имеющегося каталога сборки и его конфигурации не рекомендуется, поскольку имеющиеся настройки могут помешать работе AUH.

5. *Создание локальной конфигурации.* Нужно включить некоторые настройки в файл local.conf созданного AUH каталога сборки, как показано ниже.

   - Если нужно включить историю сборки (3.28.1. Управление историей сборки), нужно добавить в файл приведённые ниже строки.

```
INHERIT =+ "buildhistory"
BUILDHISTORY_COMMIT = "1"
```

     При такой конфигурации после успешного обновления появится файл diff в подкаталоге upgrade-helper/work/recipe/buildhistory-diff.txt каталога сборки.

   - Если нужно добавить тестирование с помощью класса testimage, следует добавить строку

```
INHERIT += "testimage"
```

     Если у дистрибутиве по умолчанию отключено свойство ptest (как в Poky) нужно добавить строку

```
DISTRO_FEATURES_append = " ptest"
```

6. *Необязательный запуск vncserver.* При работе без графического интерфейса X11 нужно запустить vncserver.

```
$ vncserver :1
$ export DISPLAY=:1
```

7. *Создание и редактирование файла конфигурации AUH.* Нужно иметь файл upgrade-helper/upgrade-helper.conf в каталоге сборки. Образец файла можно найти в репозитории AUH и воспользоваться им для создания своего файла. Например, если нужно сохранять историю сборки, следует включить её в upgrade-helper.conf.

   Если используется принятый по умолчанию файл maintainers.inc из дистрибутива Poky, размещённый в meta-yocto, не установлено maintainers_whitelist или global_maintainer_override в файле upgrade-helper.conf и задана опция -e all в командной строке AUH, утилита автоматически будет отправлять почтовые сообщения всем сопровождающим. Следует избегать этого.

Далее приведены примеры использования утилиты AUH.

- *Обновление конкретного задания* - upgrade-helper.py recipe_name. Например, для обновления xmodmap можно ввести команду upgrade-helper.py xmodmap.
- *Обновление конкретного задания до указанной версии* - upgrade-helper.py recipe_name -t version. Например, для обновления xmodmap до версии 1.2.3 служит команда upgrade-helper.py xmodmap -t 1.2.3.
- *Обновление всех заданий до последних версий без уведомления сопровождающих* - upgrade-helper.py all.
- *Обновление всех заданий до последних версий с уведомлением сопровождающих* - upgrade-helper.py -e all.

После запуска утилиты AUH можно посмотреть результаты в каталоге сборки AUH ${BUILDDIR}/upgrade-helper/timestamp. Утилита AUH также создаёт фиксации обновления заданий после успешного обновления в дереве уровня.

Можно настроить регулярный запуск AUH из задания cron. Пример этого представлен в файле weeklyjob.sh.

3.5.2. Использование devtool upgrade

Другим методом обновления заданий является команда devtool upgrade, подробно описанная в разделе Use devtool upgrade to Create a Version of the Recipe that Supports a Newer Version of the Software [2]. Справку о параметрах можно получить по команде devtool upgrade -h. Если нужно лишь узнать версию находящегося в разработке задания без его реального обновления, следует применять команду devtool latest-version recipe_name.

Как было отмечено в описании AUH, команда devtool upgrade менее автоматизирована, нежели AUH. В частности, devtool upgrade работает лишь с одним заданием, которое указано в командной строке, не может выполнять тестирование сборки и интеграции, а также автоматически создавать фиксацию (commit) изменений в дереве исходных кодов. Несмотря на эти ограничения, devtool upgrade обновляет файл задания до новой версии и пытается совместить пользовательские правки с patch-файлами репозитория. AUH на деле использует devtool upgrade, будучи своего рода «оболочкой» для devtool upgrade.

Типичный пример использования предполагает применение Git для клонирования репозитория, который будет применяться для сборки. Поскольку задание было создано раньше, уровень уже имеется в вашей конфигурации. Если же уровня нет, его можно легко добавить с помощью сценария bitbake-layers. Предположим, например, использование задания nano.bb с уровня meta-oe в репозитории meta-openembedded, также клонирование уровня в каталог /home/scottrif/meta-openembedded. Приведённая ниже команда, запущенная из каталога сборки, добавит уровень в конфигурацию сборки (${BUILDDIR}/conf/bblayers.conf):

```
$ bitbake-layers add-layer /home/scottrif/meta-openembedded/meta-oe
NOTE: Starting bitbake server...
Parsing recipes: 100% |##########################################| Time: 0:00:55
Parsing of 1431 .bb files complete (0 cached, 1431 parsed). 2040 targets, 56 skipped, 0 masked, 0 errors.
Removing 12 recipes from the x86_64 sysroot: 100% |###############| Time: 0:00:00
Removing 1 recipes from the x86_64_i586 sysroot: 100% |###########| Time: 0:00:00
Removing 5 recipes from the i586 sysroot: 100% |##################| Time: 0:00:00
Removing 5 recipes from the qemux86 sysroot: 100% |###############| Time: 0:00:00
```

Предположим, что задание nano.bb в репозитории разработки имеет версию 2.9.3, а в локальном - 2.7.4. Приведённая ниже команда, запущенная из каталога сборки, автоматически обновит задание (опция -V не обязательна и при её отсутствии devtool upgrade выполнит обновление до последней версии).

```
$ devtool upgrade nano -V 2.9.3
NOTE: Starting bitbake server...
NOTE: Creating workspace layer in /home/scottrif/poky/build/workspace
Parsing recipes: 100% |##########################################| Time: 0:00:46
Parsing of 1431 .bb files complete (0 cached, 1431 parsed). 2040 targets, 56 skipped, 0 masked, 0 errors.
NOTE: Extracting current version source...
NOTE: Resolving any missing task queue dependencies
...
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 74 tasks of which 72 didn't need to be rerun and all succeeded.
Adding changed files: 100% |#####################################| Time: 0:00:00
NOTE: Upgraded source extracted to /home/scottrif/poky/build/workspace/sources/nano
NOTE: New recipe is /home/scottrif/poky/build/workspace/recipes/nano/nano_2.9.3.bb
```

В продолжение примера используем команду devtool build для сборки обновленного задания.

```
$ devtool build nano
NOTE: Starting bitbake server...
Loading cache: 100% |#########################################################################################| Time: 0:00:01
Loaded 2040 entries from dependency cache.
Parsing recipes: 100% |#######################################################################################| Time: 0:00:00
Parsing of 1432 .bb files complete (1431 cached, 1 parsed). 2041 targets, 56 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
...
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: nano: compiling from external source tree /home/scottrif/poky/build/workspace/sources/nano
NOTE: Tasks Summary: Attempted 520 tasks of which 304 didn't need to be rerun and all succeeded.
```

В рабочем процессе devtool upgrade имеется возможность развернуть и протестировать вновь собранные программы. Однако в нашем примере команда devtool finish очищает рабочее пространство при очистке исходных кодов. Обычно это означает использование Git для подготовки и представления изменений, внесённых при обновлении. Как только дерево будет очищено, можно очистить остальное с помощью приведённой ниже команды из каталога ${BUILDDIR}/workspace/sources/nano.

```
$ devtool finish nano meta-oe
NOTE: Starting bitbake server...
Loading cache: 100% |#########################################################################################| Time: 0:00:00
Loaded 2040 entries from dependency cache.
Parsing recipes: 100% |#######################################################################################| Time: 0:00:01
Parsing of 1432 .bb files complete (1431 cached, 1 parsed). 2041 targets, 56 skipped, 0 masked, 0 errors.
NOTE: Adding new patch 0001-nano.bb-Stuff-I-changed-when-upgrading-nano.bb.patch
NOTE: Updating recipe nano_2.9.3.bb
NOTE: Removing file /home/scottrif/meta-openembedded/meta-oe/recipes-support/nano/nano_2.7.4.bb
NOTE: Moving recipe file to /home/scottrif/meta-openembedded/meta-oe/recipes-support/nano
NOTE: Leaving source tree /home/scottrif/poky/build/workspace/sources/nano as-is; if you no longer need it then please delete it manually
```

Команда devtool finish очищает рабочее пространство и создаёт patch-файл на основе фиксаций (commit). Инструмент помещает все patch-файлы обратно в дерево исходного кода (каталог nano в данном случае).

3.5.3. Обновление заданий вручную

Если по какой-то причине обновление с помощью AUH или команды devtool upgrade не подходит, можно вручную отредактировать файлы задания для обновления версии. Обновление вручную множества заданий требует значительных усилий и времени, этапы процесса кратко описаны ниже.

1. *Смена версии.* Следует переименовать задание с учётом версии (т. е. изменить PV в имени задания). Если версия не включена в имя измените значение, как это делается для PV в самом задании.

2. *Обновление* SRCREV *при необходимости.* Если исходные коды задания получены с помощью Git или иной системы контроля версий, следует обновить SRCREV с указанием хеша фиксации для новой версии.

3. *Сборка программ.* Соберите программу с помощью BitBake. Возможные отказы при сборке указаны ниже.

   - Операторы лицензирования для новой версии изменены. В этом случае следует просмотреть изменения в лицензировании и при необходимости обновить переменные LICENSE и LIC_FILES_CHKSUM. Изменения в лицензиях зачастую малозначимы, например, может измениться год в авторских правах (copyright).

   - Пользовательские изменения (patch) в старой версии могут не подойти для новой. В таких случаях нужно найти причину отказа. Исправления могут оказаться ненужными в новой версии, если проблема, вызвавшая их, была решена. Если правки нужны, но не работают, нужно изменить их для новой версии.

4. *Необязательная сборка для разной архитектуры.* После сборки новой программы для одной архитектуры можно проверить другую, изменив переменную MACHINE. Это важно для публикуемых заданий.

5. *Проверка журнала обновлений и заметок к выпускам в репозитории.* Просмотр упомянутого позволяет узнать о новых функциях и совместимости с прежними версиями. Это поможет при определении нужных действий.

6. *Необязательное создание и проверка загружаемого образа* путём загрузки на реальном устройстве.

7. *Фиксация изменений в репозитории уровня.* После успешной сборки и тестирования можно создать фиксацию (commit) для всех изменений на уровне, содержащем обновлённое задание.

3.6. Поиск временных исходных кодов

Во время разработки может представиться полезным изменить временный исходный код используемый для сборки пакетов. Например, это может потребоваться при экспериментах для внесения правок (patch). После изначальной сборки пакеты можно итеративно настроить исходный код из каталога сборки, а затем форсировать повторную компиляцию и быстрое тестирование изменённого кода. Когда решение будет найдено, можно сохранить внесённые изменения в форме правок (patch).

Во время сборки распакованный временный исходный код используется заданиями для сборки пакетов, доступный в сборочном каталоге, заданном переменной S. Ниже используется принятое по умолчанию значение S, указанное в файле meta/conf/bitbake.conf дерева исходных кодов, S = "${WORKDIR}/${BP}". Многие задания переопределяют S, например, задания, получающие исходный код с помощью Git, обычно устанавливают S = ${WORKDIR}/git.

BP представляет базовое имя задания, включающее имя и версию в форме BP = "${BPN}-${PV}". Путь к рабочему каталогу задания (WORKDIR) определяется как ${TMPDIR}/work/${MULTIMACH_TARGET_SYS}/${PN}/${EXTENDPE}${PV}-${PR}.

Реальные каталоги зависят от нескольких переменных:

- TMPDIR - выходной каталог верхнего уровня для сборки;
- MULTIMACH_TARGET_SYS - идентификатор целевой системы;
- PN - имя задания;
- EXTENDPE - эпоха (если PE не задана, что нормально для большинства заданий, значение EXTENDPE пусто);
- PV - версия задания;
- PR - ревизия (пересмотр) задания.

Предположим, например, каталог исходных кодов верхнего уровня poky, принятый по умолчанию каталог сборки poky/build и целевую систему qemux86-poky-linux. Кроме того, предположим, что задание названо foo_1.3.0.bb. В этом случае рабочим каталогом системы сборки будет poky/build/tmp/work/qemux86-poky-linux/foo/1.3.0-r0.

3.7. Использование Quilt

Quilt - это мощный инструмент для фиксации изменений исходного кода без наличия чистого дерева кода. В этом разделе рассмотрен типовой процесс, который можно применять для изменения исходного кода, тестирования изменений и их сохранения в форме patch-файлов с помощью Quilt. В части сохранения изменений при очистке задания или включении rm_work процесс devtool, описанный в [2] является более надёжным, чем использование Quilt.

Ниже приведены общие этапы работы с Quilt.

1. *Поиск исходного кода.* Временный исходный код, используемый системой сборки OE, хранится в каталоге сборки. Поиск каталога с временным кодом для конкретного пакета рассмотрен в параграфе 3.6. Поиск временных исходных кодов.

2. *Смена рабочего каталога.* Нужно перейти в каталог с временным исходным кодом, заданный переменной S.

3. *Создание правок (patch).* Перед изменением исходного кода нужно создать patch-файл с помощью команды quilt new my_changes.patch.

4. *Уведомление Quilt и добавление файлов.* После создания patch-файла нужно уведомить Quilt о файлах, которые планируется редактировать. Эти файлы могут быть добавлены к созданным правкам с помощью команды вида quilt add file1.c file2.c file3.c.

5. *Редактирование файлов.* Отредактируйте указанные ранее файлы.

6. *Тестирование изменений.* После редактирования исходного кода простейшим способом проверки внесённых изменений служит запуск задачи do_compile командой bitbake -c compile -f package. Опция -f или --force задаёт выполнение указанной в команде задачи. При возникновении проблем следует внести в исходный код соответствующие правки и повторить процедуру.

   Все изменения, внесённый во временные исходные коды, будут утеряны при запуске задачи do_clean или do_cleanall с помощью BitBake (команда bitbake -c clean package или bitbake -c cleanall package). Правки будут

утеряны также при использовании функции rm_work, как описано в параграфе 3.21. Экономия дискового пространства при сборке.

7. *Создание patch-файла.* После получения желаемых результатов правки нужно использовать Quilt для создания финального patch-файла со всеми внесёнными изменениями. Для этого служит команда quilt refresh, после выполнения которой файл my_changes.patch будет включать все правки, внесённые в файлы file1.c, file2.c и file3.c. Файл правок будет размещён в каталоге patches/ дерева исходных кодов (S).

8. *Копирование patch-файла.* Для простоты следует скопировать файл правок в каталог files, который можно создать в том же каталоге, где размещён файл задания (.bb) или добавления (.bbappend). Это позволит системе сборки OE найти файл правок. Затем patch-файл следует указать в переменной SRC_URI вашего задания, например, SRC_URI += "file://my_changes.patch".

3.8. Использование среды devshell

При отладке некоторых команд или редактировании пакетов может быть полезна оболочка devshell, при вызове которой запускаются все задачи для указанной цели, вплоть до do_patch. Затем открывается новая консоль с текущим каталогом ${S} (исходный код). В этой консоли сохраняются все переменные среды OE, относящиеся к сборке, и можно пользоваться такими командами, как configure и make. Команды выполняются как в системе сборки OE, поэтому могут быть полезны для отладки или подготовки программы к использованию в системе сборки OE. Например, для использования devshell с целью matchbox-desktop может служить команда bitbake matchbox-desktop -c devshell, которая откроет терминал в среде сборки OE. Тип командного процессора задаёт переменная OE_TERMINAL. Для терминала:

- переменная PATH включает инструменты кросс-разработки;
- переменные pkgconfig находят нужные файлы .pc;
- команда configure находит файлы сайта YP и другие нужные файлы.

В этой среде можно вызывать команды configure или compile как при работе в системе сборки OE. Рабочим каталогом служит каталог исходных кодов (S).

Для запуска вручную из devshell нужной задачи служат сценарии run.* из каталога ${WORKDIR}/temp (например, run.do_configure.pid). Если сценария для задачи нет, что бывает в случае пропуска задачи в кэше sstate, можно создать задачу за пределами devshell с помощью команды bitbake -c task.

- Выполнение сценариев run.* и выполнение задач в BitBake идентично, т. е. запуск сценария запускает задачу так же, как при использовании bitbake -c command.
- Любой сценарий run.*, не имеющий расширения .pid, является символьной ссылкой на свежую версию файла.

Механизм devshell позволяет попасть в среду выполнения задач BitBake, поэтому все команды должны вызываться так же, как это делает BitBake. Это означает предоставление соответствующих опций кросс-компиляции и т. п.

По окончании работы с devshell следует использовать команду exit или просто закрыть терминальное окно.

- При работе в devshell нужно указывать полное имя компилятора (например, arm-poky-linux-gnueabi-gcc), а не просто gcc. Это относится и к приложениям binutils, libtool и т. п. BitBake устанавливает переменные окружения, такие как CC, чтобы команда make находила нужные инструменты.
- Среда devshell поддерживает пересылку X11 и другие подобные функции.

3.9. Использование среды разработки Python

Подобно использованию devshell, описанному выше, можно работать в интерактивной среде разработки Python. При отладке некоторых команд и редактировании пакетов devpyshell может служить полезным инструментом. При вызове devpyshell запускаются все задачи для указанной цели, вплоть до do_patch в новом терминальном окне. Важные объекты и код Python доступны как при работе с задачами BitBake (в частности, хранилище данных 'd'). Ниже приведены некоторые полезные команды для работы с хранилищем данных и вызова функций.

```
pydevshell> d.getVar("STAGING_DIR")
'/media/build1/poky/build/tmp/sysroots'
pydevshell> d.getVar("STAGING_DIR")
'${TMPDIR}/sysroots'
pydevshell> d.setVar("FOO", "bar")
pydevshell> d.getVar("FOO")
'bar'
pydevshell> d.delVar("FOO")
pydevshell> d.getVar("FOO")
pydevshell> bb.build.exec_func("do_unpack", d)
pydevshell>
```

Команды выполняются как в системе сборки OE, поэтому могут применяться для отладки и подготовки программ к сборке в системе OE. Ниже приведён пример использования devpyshell для цели matchbox-desktop.

```
$ bitbake matchbox-desktop -c devpyshell
```

Эта команда открывает терминальное окно с интерактивным интерпретатором Python в среде сборки OE. Тип командного процессора задаёт переменная OE_TERMINAL. По завершении работы с devpyshell следует ввести команду exit, нажать клавиши Ctrl+d или просто закрыть терминальное окно.

3.10. Сборка

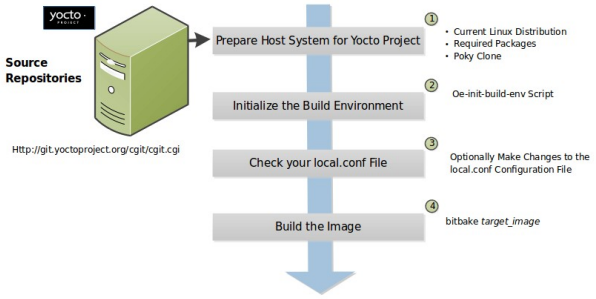
В этом разделе описаны различные процедуры сборки, включая простые варианты, целевые системы с несколькими конфигурациями, сборку образов и т. п.

3.10.1. Сборка простого образа

В среде разработки нужно собирать образ при каждом изменении поддерживаемого оборудования, добавлении или изменении системных библиотек, а также служб, с которыми связаны зависимости. Имеется несколько методов сборки образов в YP. В этом параграфе рассматриваются основные этапы сборки простого образа с использованием BitBake на сборочном хосте Linux.

- Информация о работе с интерфейсом Toaster приведена в [8].
- Сборка образов с использованием devtool описана в разделе Using devtool in Your SDK Workflow [2].
- Пример быстрой сборки образа в системе OE приведён в [4].

Процесс сборки создаёт весь дистрибутив Linux из исходных кодов и помещает его в каталог tmp/deploy/images внутри сборочного каталога. Подробное описание процесса сборки с использованием BitBake дано в разделе Images [1].



На рисунке и в приведённом ниже списке приведён обзор процесса сборки.

1. *Организация хоста для поддержки разработки с использованием YP*, как описано в главе Глава 2. Настройка YP.
2. *Инициализация среды сборки* с помощью сценария oe-init-build-env по команде source oe-init-build-env [build_dir].

   При использовании сценария инициализации система сборки OE создаёт build в качестве принятого по умолчанию каталога сборки в текущем каталоге, из которого запущен сценарий. Аргумент build_dir позволяет задать иной каталог сборки. Общепринято использовать разные каталоги сборки для каждой цели. Например, ~/build/x86 для qemux86 и ~/build/arm для qemuarm.
3. *Проверка корректности файла conf/local.conf* в каталоге с учётом реальных задач. Этот файл задаёт многие аспекты среды сборки, включая архитектуру машине в переменной MACHINE, формат пакетов для этой сборки (PACKAGE_CLASSES), и централизованный каталог загрузки архивов в переменной DL_DIR.
4. *Сборка образа* с использованием команды bitbake target [6]. Параметр target указывает имя задания для сборки. Базовыми целями являются образы из каталогов meta/recipes-core/images, meta/recipes-sato/images и т. п. в каталоге исходных кодов. Параметр target может также указывать имя задания для определённого набора программ, такого как BusyBox. Более подробные сведения о поддерживаемых системой образах даны в разделе Images [3].

   Например, по команде bitbake core-image-minimal будет собран образ core-image-minimal.

   После сборки образа его зачастую нужно установить. Образы и ядра, создаваемые системой сборки OE помещаются в каталог tmp/deploy/images внутри каталога сборки. Информация о запуске собранных образов, таких как qemux86 и qemuarm приведена в [2], установка образов рассматривается в описаниях плат и машин.

3.10.2. Сборка образов для нескольких платформ с разными конфигурациями

Можно использовать одну команду bitbake для сборки нескольких образов или пакетов для разных целей, где каждому образу или пакету нужна своя конфигурация. Такое вариант сборки называют иногда multiconfigs. В этом параграфе описана организация среды для сборки разных конфигураций и учёт зависимостей кросс-сборки для конфигураций.

3.10.2.1. Настройка и запуск сборки с несколькими конфигурациями

Для сборки нескольких конфигураций нужно задать конфигурацию для каждой цели, используя файл параллельной конфигурации в каталоге сборки, следуя требуемой иерархии файлов. Кроме того, нужно включить использование сборки нескольких конфигураций в файле local.conf.

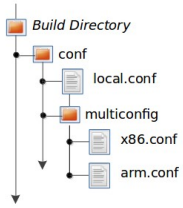
- *Создание отдельных файлов конфигурации.* Нужно создать конфигурационный файл для каждого собираемого задания. По меньшей мере каждый такой файл должен задавать машину и временный каталог, используемый BitBake для сборки. Опыт диктует создание не перекрывающихся временных каталогов для сборки, однако можно использовать общий временный каталог TMPDIR. Например, при сборке двух конфигураций для одной машины (MACHINE) qemux86 с дистрибутивами poky и poky-lsb целесообразно применять общий каталог TMPDIR.

  Ниже показаны требуемые операторы в файле конфигурации qemux86 с временным каталогом tmpmultix86.

  ```
  MACHINE="qemux86"
  TMPDIR="${TOPDIR}/tmpmultix86"
  ```

  Файлы конфигурации должны размещаться в определённом месте, а именно в подкаталоге multiconfig каталога conf. На рисунке показан пример с двумя конфигурационными файлами для x86 и arm в каталоге multiconfig.

  Использование такой иерархии обусловлено тем, что переменная BBPATH не создаётся, пока не будут проанализированы уровни. Поэтому использование заранее подготовленного файла конфигурации невозможно, если он не находится в текущем рабочем каталоге.



- *Добавление переменной BitBake для нескольких конфигураций в локальный файл конфигурации.* Переменная BBMULTICONFIG в файле conf/local.conf задаёт каждый файл конфигурации. В приведённом выше примере эта переменная имеет вид BBMULTICONFIG = "x86 arm".
- *Запуск BitBake* с помощью команды вида

  ```
  $ bitbake [multiconfig:multiconfigname:]target [[[multiconfig:multiconfigname:]target] ... ]
  ```
  Для приведённого выше примера команда будет иметь вид

  ```
  $ bitbake multiconfig:x86:core-image-minimal multiconfig:arm:core-image-sato
  ```
  В результате будет собран образ core-image-minimal, заданный файлом x86.conf, и образ core-image-sato, заданный arm.conf.

Поддержка сборки нескольких конфигураций в выпуске YP 2.7.1 (Warrior) не включает оптимизации общего состояния (sstate), поэтому при использовании одного объекта несколько раз создаётся несколько каталогов TMPDIR и сборка использует имеющийся кэш sstate или запускается заново.

3.10.2.2. Включение зависимостей при сборке нескольких конфигураций

Иногда могут возникать зависимости между целями (multiconfig) при сборке нескольких конфигураций. Предположим, например, что при сборке образа core-image-sato для x86 нужна корневая файловая система arm. Это может быть связано с тем, что задача do_image в core-image-sato зависит от выполнения задачи do_rootfs в core-image-minimal. Для включения зависимостей при сборке нескольких конфигураций их нужно объявить в задании как task_or_package[mcdepends] = "multiconfig:from_multiconfig:to_multiconfig:recipe_name:task_on_which_to_depend".

Рассмотрим в качестве примера конфигурацию из предыдущего параграфа. В этом случае нужно в задание для образа core-image-sato добавить строку do_image[mcdepends] = "multiconfig:x86:arm:core-image-minimal:do_rootfs". В этом примере в качестве from_multiconfig служит x86, а в качестве to_multiconfig - arm. Задачей, от которой зависит do_image, является do_rootfs из задания core-image-minimal, связанного с arm.

После указания зависимости можно собрать образ x86 с помощью команды bitbake multiconfig:x86:core-image-sato. Команда выполняет все задачи, требуемые при создании образа core-image-sato для x86. В результате задания зависимости BitBake выполняет также задачу do_rootfs для arm.

Наличие задания, зависимого от корневой файловой системы другого задания может показаться бессмысленным. Рассмотрим это на примере оператора в задании core-image-sato.

```
do_image[mcdepends] = "multiconfig:x86:arm:core-image-minimal:do_image"
```
В этом случае BitBake нужно создать образ core-image-minimal для сборки arm, поскольку от него зависит сборка x86. В результате того, что x86 и arm включены во множественную конфигурацию и имеют раздельные конфигурационные файлы, BitBake помещает результаты сборки в соответствующие временные каталоги (TMPDIR).

3.10.3. Сборка образа initramfs

Образ первичного RAM-диска (initramfs) обеспечивает временную корневую файловую систему на раннем этапе инициализации (например, загрузка модулей, требуемых для нахождения и монтирования корневой системы).

Образ initramfs является «наследником» initrd и использует архив cpio на исходной файловой системе для загрузки в память при запуске Linux. Поскольку Linux использует содержимое архива в процессе инициализации, в образ initramfs нужно включать все драйверы и инструменты, нужные для монтирования окончательной корневой файловой системы.

Ниже перечислены этапы создания образа initramfs.

1. *Подготовка задания для initramfs.* Можно воспользоваться заданием core-image-minimal-initramfs.bb из каталога meta/recipes-core в дереве исходных кодов как примером.
2. *Решение вопроса о встраивании образа initramfs в образ ядра.* Если нужно объединить образ initramfs с образом ядра, следует задать INITRAMFS_IMAGE_BUNDLE = "1" в файле local.conf и установить переменную INITRAMFS_IMAGE в задании для сборки ядра. Рекомендуется объединять образы ядра и initramfs для предотвращения зацикливания зависимостей при включении в образ initramfs модулей ядра.

   Установка флага INITRAMFS_IMAGE_BUNDLE ведёт к распаковке образа initramfs в каталог ${B}/usr/. Распакованный образ initramfs передаётся в Makefile ядра через переменную CONFIG_INITRAMFS_SOURCE, что позволяет встроить образ initramfs в ядро.

   Если образ initramfs не встраивается в образ ядра, это означает использование initrd. Создание initrd выполняется в основном через переменные INITRD_IMAGE, INITRD_LIVE и INITRD_IMAGE_LIVE (см. описание image-live.bbclass).
3. *Добавление элементов в initramfs через задания.* При добавлении элементов initramfs следует использовать переменную PACKAGE_INSTALL, а не IMAGE_INSTALL, поскольку это обеспечивает более прямой контроль дополнений по сравнению с принятыми по умолчанию настройками классов image и core-image.
4. *Сборка образов ядра и initramfs.* Ядро собирается с использованием BitBake. Поскольку задание для ядра зависит от initramfs, образ initramfs собирается и связывается с ядром, если установлена переменная INITRAMFS_IMAGE_BUNDLE, как указано выше.

3.10.4. Сборка миниатюрной системы

Компактные дистрибутивы могут обеспечивать ряд преимуществ, таких как меньшие потребности в памяти (снижение цены), повышение производительности за счёт эффективного кэширования, снижение энергопотребления за счёт уменьшения размера памяти, ускорение загрузки и снижение расходов на разработку. Примерами таких систем в реальном мире являются цифровые камеры, медицинское оборудование и т. п.

В этом параграфе приведена информация о способах сокращения размера дистрибутива по сравнению с poky-tiny, занимающим около 5 Мегабайт.

3.10.4.1. Обзор

Ниже представлены рассмотренные в соответствующих параграфах этапы снижения размера корневой файловой системы для обеспечения быстрой загрузки в сочетании с требуемой функциональностью без использования начального RAM-диска.

- 3.10.4.2. Цели и принципы
- 3.10.4.3. Влияние компонент на размер образа
- 3.10.4.4. Минимизация корневой файловой системы
- 3.10.4.5. Минимизация ядра
- 3.10.4.6. Исключение требований к управлению пакетами
- 3.10.4.7. Другие способы снижения размера
- 3.10.4.8. Итерации процесса

3.10.4.2. Цели и принципы

Ниже приведён список вопросов, которые нужно решить при создании компактных дистрибутивов:

- определение требуемого размера (например, ядро меньше 1 Мбайт и корневая файловая система не больше 3 Мбайт);
- определение областей, занимающих большую часть пространства для концентрации усилий по сокращению;
- отказ от внесения сложных изменений для достижения цели;
- использование специфичных для устройства опций;
- работа с отдельным уровнем для изоляции изменений (см. раздел 3.1. Уровни и их создание).

3.10.4.3. Влияние компонент на размер образа

Проще всего начать с создания своего дистрибутива. Можно воспользоваться в качестве примера имеющимся в YP дистрибутивом poky-tiny, для чего следует установить для переменной DISTRO в файле local.conf значение "poky-tiny", как описано в разделе 3.19. Создание своего дистрибутива.

Понимание концепций работы с памятью позволит уменьшить размер системы. Память включает статические, динамические и временные разделы. Статическая память - это разделы TEXT (код), DATA (инициализированные данные в коде) и BSS (неинициализированные данные). Динамическая память выделяется в процессе работы для стека, хэш-таблиц и т. п. Временная память включает области для распаковки ядра и функций __init__ и освобождается по завершении загрузки.

Для просмотра текущих размеров ядра и корневой файловой системы в каталоге scripts/tiny/ дерева исходных кодов есть два полезных сценария:

- *ksize.py* определяет размер компонент собранного ядра;
- *dirsize.py* определяет размер компонент корневой файловой системы.

Ещё один сценарий и команда помогают организовать фрагменты конфигурации и увидеть зависимости файлов.

- *merge_config.sh* помогает управлять конфигурационными файлами и фрагментами конфигурации ядра. Сценарий позволяет объединять фрагменты конфигурации, а также создавать переопределения и выдаёт предупреждения о пропущенных опциях. Это хорошо подходит для итеративной настройки конфигурации и создания конфигурационных файлов для разных машин без дублирования процессов.

  Сценарий является частью репозитория Linux Yocto (linux-yocto-3.14, linux-yocto-3.10, linux-yocto-3.8 и т. п.) и хранится в каталоге scripts/kconfig. Дополнительная информация о фрагментах конфигурации приведена в разделе Creating Configuration Fragments [7].

- *bitbake -u taskexp -g bitbake_target* открывает Dependency Explorer для просмотра зависимостей. Понимание зависимостей позволяет принять обоснованные решения при исключении компонент ядра и корневой файловой системы.

3.10.4.4. Минимизация корневой файловой системы

Корневая файловая система содержит пакеты для загрузки, библиотеки и приложения. Для изменения размера можно поменять способ подготовки пакетов, а также саму файловую систему.

Сначала следует разобраться с размерами файловой системы и отдельных файлов с помощью сценария dirsize.py:

```
$ cd root-directory-of-image
$ dirsize.py 100000 > dirsize-100k.log
$ cat dirsize-100k.log
```

Параметр 100000 задаёт игнорирование файлов, размер которых меньше 100 кбайт. Сценарий возвращает размеры несжатых файлов, поэтому на файловых системах с компрессией нужно вводить коэффициент. После просмотра журнала dirsize-100k.log можно будет сосредоточиться на крупных файлах.

Следует учитывать взаимосвязи между файлами, чтобы не навредить функциональности. Можно посмотреть зависимости с помощью интерфейса Dependency Explorer пакета BitBake, как показано ниже.

```
$ cd image-directory
$ bitbake -u taskexp -g image
```

Изучив зависимости можно понять, какие пакеты или файлы можно безбоязненно исключить. При выборе пакетов для удаления следует учитывать их функциональность. Например, может оказаться ненужным дисплей VGA или можно обойтись devtmpfs и mdev вместо udev. Изменения следует вносить в файл local.conf. Например, для исключения udev и glib можно указать VIRTUAL-RUNTIME_dev_manager = "".

Следует также учитывать тип корневой файловой системы и её соответствие поставленным задачам. Например, можно рассмотреть системы cramfs, squashfs, ubifs, ext2 или initramfs. Следует учитывать, что файловой системе ext3 потребуется 1 Мбайт для журнала, который не будет нужен в файловой системе с доступом лишь для чтения.

После каждого исключения следует заново собирать систему, контролируя функциональность и размер.

3.10.4.5. Минимизация ядра

Ядро собирается с включением правил для зависимых от оборудования аспектов, включая разрешённые подсистемы, архитектуру, драйверы. При необходимости может изменяться исходный код ядра.

Сценарий ksize.py из каталога сборки верхнего уровня может подсказать, что следует изменить в ядре.

```
$ cd top-level-linux-build-directory
$ ksize.py > ksize.log
$ cat ksize.log
```

При просмотре журнала можно увидеть размер встроенных файлов .o для драйверов, сети, базовых функций, файловых систем, звука и т. п. Сценарий возвращает размеры несжатых компонент и для сжатых ядер следует вводить коэффициент. Следует выделить наиболее крупные области и оптимизировать их размеры. Для более подробного изучения структуры служит специальная опция -d.

```
$ ksize.py -d > ksize.log
```

Просмотр журнала позволяет определить компоненты, которые можно исключить. Например, если в системе нет звуковых устройств, можно исключить драйверы для звука.

После определения исключаемых компонент нужно заново настроить конфигурацию ядра для учёта изменений при следующей сборке. Это можно сделать с помощью команды menuconfig. Однако в этом случае могут возникнуть затруднения с учётом влияния отдельных исключений, а также с репликацией изменений на другие целевые устройства. Лучше начать с пустой конфигурации, введя команду allnoconfig, создать фрагменты конфигурации для отдельных изменений, а затем использовать сценарий merge_config.sh. Это упрощает итерации настройки конфигурации в цикле сборки.

При каждом изменении конфигурации следует выполнять сборку ядра для контроля размера и результатов изменений.

3.10.4.6. Исключение требований к управлению пакетами

Требования управления пакетами ведут к увеличению образа, поэтому исключение этих требований позволяет сделать образ более компактным. Это включает удаление менеджера пакетов с его зависимостями, а также всех данных управления пакетами.

Для исключения требований к управлению пакетами следует убедиться в отсутствии значения package-management в операторе IMAGE_FEATURES для образа. При удалении этого свойства из корневой файловой системы будет исключён менеджер пакетов и все его зависимости.

3.10.4.7. Другие способы снижения размера

В зависимости от конкретных задач могут быть и иные способы снижения размера. При поиске решений могут помочь указанные здесь инструменты в сочетании с экспериментами и итерациями.

- *glibc*
  - удалите из DISTRO_FEATURES свойства glibc, которые представляются ненужными;
  - соберите дистрибутив;
  - при возникновении отказа по причине отсутствия символов в пакете определите возможность настройки пакета без этих символов;
  - повторите сборку и проверку работоспособности.
- *busybox. Процедуры похожи на описанные выше для* glibc. Разница заключается в необходимости загрузки получившейся системы для проверки. Нужно обеспечить интеграцию фрагментов конфигурации в Busybox, поскольку BusyBox самостоятельно обрабатывает основные возможности, позволяя добавлять фрагменты конфигурации.

3.10.4.8. Итерации процесса

Если цель не достигнута с первой попытки, процесс следует повторить. Следует обращать внимание на компоненты, занимающие 90% корневой файловой системы и ядра. Удаляйте компоненты, которые не нужны при работе системы.

В зависимости от конкретной системы следует обращать внимание на Busybox, где обеспечиваются урезанные версии инструментов Unix в одном исполняемом файле.

3.10.5. Сборка образов для нескольких машин

Разработчикам зачастую приходится создавать образы разных машин, использующих одинаковую программную среду. В таких случаях заманчиво установить настройки и флаги оптимизации для каждой сборки специально под целевую платформу. Однако это может существенно увеличить время сборки и осложнить поддержку хранилища пакетов для всех машин. Например, выбор настроек, наиболее подходящих для процессора платформы, может включать тонкую оптимизацию GCC для определённой платформы, но без этого не будет достигнуто существенного роста производительности по сравнению с применением базовых настроек для всех платформ. Вместо этого можно выполнить настройки, которые заставят систему сборки OE многократно использовать программные компоненты на разных платформах, когда это имеет смысл.

Если скорость сборки и обслуживание хранилища пакетов важны, следует рассмотреть приведённые ниже рекомендации.

- *Общий каталог сборки.* По возможности следует применять для сборок общий каталог TMPDIR. YP поддерживает переключение между значениями MACHINE в одном каталоге TMPDIR. Такая практика хорошо поддерживается и часто применяется разработчиками при сборке для нескольких машин. При использовании общего каталога TMPDIR система сборки OE может многократно применять имеющиеся естественные и кросс-задания для разных машин, что снижает время сборки.

  При изменении настроек DISTRO или важных параметров конфигурации (например, схема файловой системы) нужно начинать работу с чистого каталога TMPDIR. Общий каталог TMPDIR в таких случаях может работать, но это не гарантируется.

- *Включение подходящей архитектуры управления пакетами.* По умолчанию система сборки OE включает 3 уровня управления пакетами - all, tune или package, machine. Задания обычно выбирают для своего вывода одну архитектуру (тип). В зависимости от цели создания пакетов заданием наличие соответствующей архитектуры управления пакетами может напрямую влиять на время сборки.

  Задание, создающее сценарии, может выбрать архитектуру all, поскольку оно не создаёт двоичных файлов. Для выбора этой архитектуры следует обеспечить наследование класса allarch, который в данном случае настраивает множество переменных так, что пакеты подходят для разной архитектуры.

  Если задание создаёт машинозависимые пакеты или одна из зависимостей при работе или сборке уже включает машину или архитектуру, что делает задание машинозависимым, следует выбирать тип machine через переменную MACHINE_ARCH в виде PACKAGE_ARCH = "${MACHINE_ARCH}".

  Если архитектура управления пакетами осознанно не включена через PACKAGE_ARCH, система сборки OE по умолчанию устанавливает PACKAGE_ARCH = "${TUNE_PKGARCH}".

- *Выбор базового файла настройки (по возможности).* Некоторые настройки являются достаточно общими и могут применяться для разных платформ (например, armv5 устанавливает пакеты, которые в большинстве случаев могут работать на armv6 и armv7, а двоичные файлы i486 могут работать на i586 и последующих платформах). Однако следует понимать, что усовершенствования новых процессоров работать не будут.

  При выборе одной настройки для множества машин система сборки OE повторно использует собранные ранее программы, что сокращает время сборки. Даже при смене sysroot для машин программы не компилируются заново и в хранилище существует лишь один пакет.

- *Управление детализацией подготовки пакетов.* Иногда бывает полезно внедрение иного уровня архитектуры пакетов в дополнение к 3 указанным ранее. В качестве примера рассмотрим, как NXP (ранее Freescale) позволяет повторно использовать двоичные пакеты на своём уровне meta-freescale. Класс fsl-dynamic-packagearch обеспечивает совместное использование пакетов GPU для плат i.MX53, поскольку на всех применяется AMD GPU. Для плат i.MX6 можно поступить аналогично, поскольку все они используют Vivante GPU. Этот класс проверяет хранилище данных BitBake на предмет наличия пакетов, предоставляющих такую субархитектуру или зависящих от неё. При наличии таких пакетов класс устанавливает значение PACKAGE_ARCH на основе значения MACHINE_SUBARCH. Если пакет не предоставляет такую субархитектуру и не зависит от неё., но соответствует значению специфичного для машины фильтра, устанавливается MACHINE_ARCH. Это снижает число собираемых пакетов и экономит время.

- *Использование инструментов отладки.* Иногда возникают случаи повторной сборки программ без вашего ведома. Например, система сборки OE может не использовать общего состояния с другой машиной, которое вы предполагаете. Такие ситуации часто возникают в результате ссылок на машинозависимые переменные, такие как MACHINE, SERIAL_CONSOLES, XSERVER, MACHINE_FEATURES и т. п. в коде, который должен зависеть лишь от настройки или когда задание зависит (DEPENDS, RDEPENDS, RRECOMMENDS, RSUGGESTS и т. п.) от другого задания, уже определившего PACKAGE_ARCH как "${MACHINE_ARCH}".

  Для таких случаев есть инструменты, помогающие разобраться в ситуации.

  - *sstate-diff-machines.sh* в каталоге scripts репозитория исходных кодов. Информация о работе со сценарием приведена в комментариях.

  - *Опция BitBake "-S printdiff"* заставляет BitBake попытаться насти максимальное соответствие подписи (например, в кэше общего состояния) и запустить bitbake-diffsigs для определения штампов и приращений (delta), где два штампа разошлись.

3.10.6. Сборка программ из внешних источников

По умолчанию система сборки OE использует в процессе работы сборочный каталог. Процесс сборки включает извлечение файлов исходного кода, их распаковку и применение исправлений (если они есть). В некоторых случаях может потребоваться собрать программы из источников, расположенных вне системы сборки OE. Например, проект может включать BSP, ядро для которого имеет весьма специальные настройки, и при этом нужно минимизировать работу команды разработчиков с системой сборки, чтобы они могли сосредоточиться на своём проекте. В этом случае

специализированное ядро может размещаться на машине, где осуществляется разработка и в переменной SRC_URI задания для сборки будет указан внешний каталог, чтобы не копировать файлы.

Для сборки программы из внешнего источника нужно наследовать класс externalsrc и указать в переменной EXTERNALSRC местоположение исходного кода. Эти операторы помещаются в файл local.conf, как показано ниже.

```
INHERIT += "externalsrc"
EXTERNALSRC_pn-myrecipe = "path-to-your-source-tree"
```

Можно также задать переменные в файле задания или файле дополнения. Например,

```
EXTERNALSRC = "path"
EXTERNALSRC_BUILD = "path"
```

Чтобы эти установки сработали, нужно глобально или локально наследовать класс externalsrc.

По умолчанию класс externalsrc.bbclass собирает исходный код в каталоге, отличающемся от внешнего каталога источников, заданного в EXTERNALSRC. Если нужно собрать исходный код в том же каталоге, где он хранится или в неком ином названном каталоге, можно указать нужный каталог в переменной EXTERNALSRC_BUILD

```
EXTERNALSRC_BUILD_pn-myrecipe = "path-to-your-source-tree"
```

3.10.7. Автономная репликация сборки

Иногда полезно сделать «моментальный снимок» разрабатываемого кода, использованного для сборки и затем применять этот снимок для репликации сборки в автономном режиме. Для этого нужно сначала подготовить и заполнить каталог файлами «снимка». После загрузки файлов их можно применять для репликации сборки в любой момент и с любой машины. Процедура подготовки описана ниже.

1. *Создание пустого каталога для загрузки* (DL_DIR). Можно очистить имеющийся каталог или указать новое значение в переменной DL_DIR.
2. *Генерация архивов репозитория Git.* В файле local.conf нужно указать строки:

   ```
   DL_DIR = "/home/your-download-dir/"
   BB_GENERATE_MIRROR_TARBALLS = "1"
   ```

   При выборке файлов на следующем этапе BitBake соберёт файлы исходного кода и создаст архив в каталоге DL_DIR. Дополнительная информация приведена в описании BB_GENERATE_MIRROR_TARBALLS.
3. *Заполнение каталога загрузки без сборки* с помощью BitBake:

   ```
   $ bitbake target --runonly=fetch
   ```

   После этого каталог загрузки (${DL_DIR}) будет содержать снимок исходного кода в виде архива.
4. *Необязательное удаление служебных данных Git или иных SCM.* Можно удалить из каталога загрузки подкаталоги Git или других SCM, такие как ${DL_DIR}/git2/*, поскольку они присутствуют в архиве.

После заполнения каталога загрузки можно создать «своё зеркало» и собрать программу для своей целевой платформы, как описано ниже.

1. *Использование только локальных файлов.* В файл local.conf следует добавить переменную SOURCE_MIRROR_URL, наследование класса own-mirrors и переменную BB_NO_NETWORK.

   ```
   SOURCE_MIRROR_URL ?= "file:///home/your-download-dir/"
   INHERIT += "own-mirrors"
   BB_NO_NETWORK = "1"
   ```

   SOURCE_MIRROR_URL и класс own-mirror задают использование каталога загрузки в качестве «своего зеркала», а переменная BB_NO_NETWORK заставляет BitBake извлекать файлы из локального источника.
2. *Начало с «чистого листа»* путём удаления каталога ${TMPDIR} или использования нового каталога сборки.
3. *Сборка цели* с использованием BitBake

   ```
   $ bitbake target
   ```

   Сборка выполняется с использованием локального снимка исходных файлов. Автономная сборка не будет работать, если задание пытается найти свежую версию программ путём установки SRCREV = "${AUTOREV}", поскольку в этом случае система обращается в сеть, пытаясь найти последнюю версию программы в SCM. Задания, использующие AUTOREV, обычно являются пользовательскими или изменёнными, а задания из публичных репозиториев как правило не применяют AUTOREV.

   При наличии заданий с AUTOREV можно выполнить приведённые ниже рекомендации для автономной сборки.

   a. Использовать конфигурацию, созданную включением истории сборки (3.28.1. Управление историей сборки).

   b. Использовать команду buildhistory-collect-srcrevs для сбора сохранённых значений SRCREV из истории сборки (3.28.2.1. История сборки пакета).

   c. Поскольку имеются корректные выпуски исходного кода, можно изменить задания, установив в SRCREV нужные версии программ.

3.11. Ускорение сборки

Время сборки может быть продолжительным. По умолчанию система сборки применяет простые элементы управления для повышения эффективности сборки. Обычно принятые по умолчанию значения перечисленных ниже переменных обеспечивают наиболее эффективную сборку на системах с одним процессором (не ядром). Если на хосте имеется несколько CPU, можно попытаться изменить эти переменные для повышения скорости сборки.

- BB_NUMBER_THREADS задаёт максимальное число потоков для выполнения задач BitBake.
- BB_NUMBER_PARSE_THREADS задаёт число потоков, используемых BitBake при анализе заданий.

- PARALLEL_MAKE - добавочные опции, передаваемые команде make в задаче do_compile для выполнения параллельной компиляции на локальном хосте сборки.
- PARALLEL_MAKEINST - добавочные опции, передаваемые команде make в задаче do_install для выполнения параллельной инсталляции на локальном хосте сборки.

Эти переменные управляют числом процессорных ядер, доступных системе сборки. Для однопроцессорных систем автоматическое масштабирование обеспечивает использование системой сборки возможностей параллельного выполнения операций. Ниже рассмотрены дополнительные факторы, способные влиять на скорость сборки.

- *Тип файловой системы* на хосте сборки может влиять на скорость. Рекомендуется использовать файловые системы ext4, поскольку они обеспечивают более высокую производительность по сравнению с ext2 и ext3.
- *Запрет обновления времени доступа с помощью noatime*. Опция монтирования noatime позволяет системе сборки не обновлять время доступа к файлам и каталогам.
- *Более продолжительный интервал фиксации*. Использовании опции монтирования "commit=" позволяет задать интервал (в секундах) между записями дискового кэша. Смена принятого по умолчанию интервала в 5 секунд на более долгий повышает производительность сборки, но также увеличивает риск потери данных.
- *Выбор системы управления пакетами*. Из числа доступных менеджеров самым быстрым является IPK. Кроме того, ускоряет сборку использование единственного менеджера вместо нескольких.
- *Использование файловой системы tmpfs для каталога TMPDIR* может ускорить сборку, но преимущества ограничены, поскольку компилятор использует -pipe. Система сборки предпринимает несколько действий по предотвращению вызовов sync() для файловой системы, исходя из того, что при серьёзном отказе содержимое каталога сборки легко создать заново.
- *Наследование класса rm_work* позволяет ускорить сборку за счёт снижения объёма данных, сохраняемых в кэше и на диске. Это также ускоряет очистку TMPDIR. Разработчики файловых систем рекомендуют в качестве наиболее быстрого способа удаления большого числа файлов переформатирование файловой системы. Однако это требует соответствующей организации дисковых разделов.

В дополнение к перечисленным рекомендациям отметим несколько компромиссов, которые могут ускорить сборку.

- *Удаление ненужных элементов* из DISTRO_FEATURES.
- *Исключение отладочных символов и других данных отладки* путём запрета генерации пакетов*-dbg установкой INHIBIT_PACKAGE_DEBUG_SPLIT = "1".
- *Запрет создания статических библиотек для заданий, выведенных из autoconf или libtool*, как показано ниже.

```
STATICLIBCONF = "--disable-static"
STATICLIBCONF_sqlite3-native = ""
EXTRA_OECONF += "${STATICLIBCONF}"
```

  - Некоторым заданиям статические библиотеки нужны для корректной работы (например, pseudo-native требует sqlite3-native). Приведённые выше переопределения учитывают это.
  - Для подготовки некоторых пакетов нужны статические библиотеки и может потребоваться их сборка.

3.12. Работа с библиотеками

Библиотеки являются важной частью сборки и здесь даны некоторые рекомендации по работе с библиотеками.

3.12.1. Включение файлов статических библиотек

При создании библиотеки, предлагающей статическую компоновку, можно управлять включением в собираемую библиотеку статических файлов (*.a). Переменные PACKAGES и FILES_* в файле meta/conf/bitbake.conf определяют включение в пакеты файлов, установленных задачей do_install. По молчанию PACKAGES включает ${PN}-staticdev, представляя все файлы статических библиотек. В нескольких прежних выпусках YP статические библиотеки указывались как ${PN}-dev. Ниже приведён фрагмент файла конфигурации BitBake для файлов статических библиотек.

```
PACKAGE_BEFORE_PN ?= ""
PACKAGES = "${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}"
PACKAGES_DYNAMIC = "^${PN}-locale-.*"
FILES = ""

FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* ${libdir}/lib*${SOLIBS} \
${sysconfdir} ${sharedstatedir} ${localstatedir} \
${base_bindir}/* ${base_sbindir}/* \
${base_libdir}/*${SOLIBS} \
${base_prefix}/lib/udev/rules.d ${prefix}/lib/udev/rules.d \
${datadir}/${BPN} ${libdir}/${BPN}/* \
${datadir}/pixmaps ${datadir}/applications \
${datadir}/idl ${datadir}/omf ${datadir}/sounds \
${libdir}/bonobo/servers"

FILES_${PN}-bin = "${bindir}/* ${sbindir}/*"

FILES_${PN}-doc = "${docdir} ${mandir} ${infodir} ${datadir}/gtk-doc \
${datadir}/gnome/help"
SECTION_${PN}-doc = "doc"

FILES_SOLIBSDEV ?= "${base_libdir}/lib*${SOLIBSDEV} ${libdir}/lib*${SOLIBSDEV}"
FILES_${PN}-dev = "${includedir} ${FILES_SOLIBSDEV} ${libdir}/*.la \
${libdir}/*.o ${libdir}/pkgconfig ${datadir}/pkgconfig \
${datadir}/aclocal ${base_libdir}/*.o \
```

```
${libdir}/${BPN}/*.la ${base_libdir}/*.la"
SECTION_${PN}-dev = "devel"
ALLOW_EMPTY_${PN}-dev = "1"
RDEPENDS_${PN}-dev = "${PN} (= ${EXTENDPKGV})"

FILES_${PN}-staticdev = "${libdir}/*.a ${base_libdir}/*.a ${libdir}/${BPN}/*.a"
SECTION_${PN}-staticdev = "devel"
RDEPENDS_${PN}-staticdev = "${PN}-dev (= ${EXTENDPKGV})"
```

3.12.2. Объединение разных версий библиотек в одном образе

Система сборки позволяет создавать библиотеки с оптимизацией под разные платформы и архитектуру, объединяя их в одном образе. Можно связать разные исполняемые файлы образа с разными библиотеками для конкретных вариантов применения. Это свойство называется Multilib. Примером может служить ситуация, когда большая часть системы собрана в 32-битовом режиме с 32-битовыми библиотеками, но есть некоторые 64-битовые приложения, которым нужны 64-битовые библиотеки. Хотя feature чаще всего используется для обслуживания различий между 32- и 64-битовыми приложениями, система сборки упрощает и другие варианты оптимизации. Можно собрать некоторые приложения для работы с одним набором библиотек, другие - с иным. Библиотеки могут различаться архитектурой, опциями компиляции и другими параметрами оптимизации. Несколько примеров представлено на уровне meta-skeleton:

- conf/multilib-example.conf;
- conf/multilib-example2.conf;
- recipes-multilib/images/core-image-multilib-example.bb.

3.12.2.1. Подготовка к использованию Multilib

Применение Multilib обусловлено потребностями пользователей, поэтому нет готовой конфигурации на все случаи. Для включения Multilib нужно сначала обеспечить поддержку разных библиотек в задании, что уже реализовано во многих стандартных заданиях. Следует проверить значение переменной BBCLASSEXTEND в файле meta/conf/multilib.conf дерева источников. В конечном итоге поддержка будет включена во все задания и проверка станет ненужной.

Расширение Multilib в основном работает автоматически, преобразуя имя пакета ${PN} в ${MLPREFIX}${PN}, где MLPREFIX указывает конкретную библиотеку (например, "lib32-" или "lib64-"). Стандартные переменные, такие как DEPENDS, RDEPENDS, RPROVIDES, RRECOMMENDS, PACKAGES и PACKAGES_DYNAMIC преобразуются системой автоматически. Если задание преобразуется вручную, можно использовать переменную ${MLPREFIX} для корректного преобразования имён. Код автоматического преобразования включён в класс multilib.bbclass.

3.12.2.2. Использование Multilib

После настройки заданий нужно указать требуемую комбинацию библиотек в файле local.conf каталога сборки.

```
MACHINE = "qemux86-64"
require conf/multilib.conf
MULTILIBS = "multilib:lib32"
DEFAULTTUNE_virtclass-multilib-lib32 = "x86"
IMAGE_INSTALL_append = " lib32-glib-2.0"
```

В приведённом примере включена дополнительная библиотека lib32. При объединении с вариантами lib32 в примере используется x86. Эту настройку можно посмотреть в файле meta/conf/machine/include/ia32/arch-ia32.inc. Пример включает lib32-glib-2.0 во все образы, что является одним из вариантов. Можно использовать обычную сборку образа для включения этой зависимости. Например, bitbake core-image-sato. Можно также собрать пакеты Multilib независимо с помощью команды вида bitbake lib32-glib-2.0.

3.12.2.3. Детали реализации

Имеются как общие, так и связанные с системой управления пакетами детали реализации. Рассмотрим сначала общие.

- Типичное соглашения для кода расширения класса, используемого Multilib, предполагает, что все имена пакетов из PACKAGES, включающие ${PN}, начинаются с ${PN}. При указании ${PN} не в начале имени возникают проблемы.
- Значение TARGET_VENDOR в Multilib расширено до "-vendormlmultilib" (например, -pokymllib32 для lib32 в Multilib с Poky). Причиной послужило то, что символы - в строке производителя в настоящее время противоречат config.sub в Autoconf, а использование иных разделителей проблематично.

Для системы управления пакетами RPM также существует ряд деталей, указанных ниже.

- Определённая отдельная архитектура для пакетов Multilib, а также отдельный каталог развёртывания tmp/deploy/rpm. Например, для lib32 в образе qemux86-64 будут представлены архитектуры "all", "qemux86_64", "lib32_qemux86_64" и "lib32_x86".
- Переменная ${MLPREFIX} вырезается из ${PN} при подготовке пакетов RPM. Именование обычных и Multilib пакетов RPM в системе qemux86-64 будет иметь вид bash-4.1-r2.x86_64.rpm и bash-4.1.r2.lib32_x86.rpm.
- Для образа Multilib менеджер RPM сначала устанавливает базовый образ, затем - библиотеки Multilib.
- Система сборки использует RPM при совпадении имён. файлов в нескольких пакетах Multilib.

Детали, связанные с системой управления IPK, перечислены ниже.

- ${MLPREFIX} не удаляется из ${PN} при подготовке пакетов IPK. Именование обычных и Multilib пакетов IPK в системе qemux86-64 имеет вид bash_4.1-r2.x86_64.ipk и lib32-bash_4.1-rw_x86.ipk.
- Каталог развёртывания IPK не меняется при использовании ${MLPREFIX}, поскольку пакеты различаются значениями ${PN}.

- IPK проверяет пригодность установки Multilib с использованием правил сравнения, переопределения и т. п.

3.12.3. Установка нескольких версий одной библиотеки

Может возникать потребность использовать одновременно несколько версий библиотеки в одной системе. Это почти всегда происходит при смене версии API, если имеется несколько программ, работающих с разными версиями библиотеки. Решением проблемы является параллельная установка нескольких версий в одной системе. Сделать это достаточно просто, если библиотеки корректно управляют версиями. Нужно просто отдельно указать библиотеки путём подготовки заданий с именами, где переменная PN включает версию библиотеки (например, старшую часть номера). В результате каждое задание будет загружать свою версию библиотеки. Например, показанные ниже примеры имён. заданий для библиотеки clutter позволяют использовать две версии.

```
clutter-1.6_1.6.20.bb
clutter-1.8_1.8.4.bb
```

Если другие задания зависят от конкретной версии библиотеки, нужно указать соответствующую версию в переменной DEPENDS таких заданий. Например, для задания, зависящего от библиотеки clutter версии 1.8, следует указать DEPENDS = "clutter-1.8".

3.13. Использование x32 psABI

Двоичный интерфейс приложений x32 ABI (x32 psABI) является естественным ABI для архитектуры Intel® 64 (x86-64) и определяет соглашения о вызовах функций в среде обработки, задавая используемые регистры и размеры различных типов данных C.

Некоторые среды обработки предпочитают использовать 32-битовые приложения даже на 64-разрядных платформах Intel. Рассмотрим i386 psABI, который является самым старым из 32-битовых ABI для платформ Intel 64. i386 psABI не обеспечивает эффективного использования и доступа ко всем ресурсам 64-битовых процессоров Intel 64, оставляя процессоры недогруженными. По сравнению с этим интерфейс x86_64 psABI более новый и использует 64 бита для размеров данных и указателей. Дополнительные биты увеличивают размер программ и библиотек, а также потребности в памяти и дисковом пространстве. Работа в среде x32 psABI позволяет позволяет программам более эффективно использовать ресурсы CPU и системы, а также снижать размер приложений. Дополнительные биты используются для регистров, но не для адресации.

YP поддерживает финальную спецификацию x32 psABI, как показано ниже.

- Создание пакетов и образов в формате x32 psABI для целевых платформ x86_64.
- Возможность собирать задания, с использованием инструментария x32.
- Возможность создавать и загружать образы core-image-minimal и core-image-sato images.
- Поддержка менеджером RPM имеющихся двоичных файлов x32.
- Поддержка больших образов.

Для использования x32 psABI нужно включить в файл conf/local.conf приведённые ниже строки.

```
MACHINE = "qemux86-64"
DEFAULTTUNE = "x86-64-x32"
baselib = "${@d.getVar('BASE_LIB_tune-' + (d.getVar('DEFAULTTUNE') \
   or 'INVALID')) or 'lib'}"
```

После этого можно применять BitBake для сборки образов с поддержкой x32 psABI, например. bitbake core-image-sato.

3.14. Включение поддержки GObject Introspection

GObject introspection является стандартным механизмом доступа к программам на основе GObject из рабочей среды. GObject является свойством библиотеки Glib, которое предоставляет объектную среду на базе GNOME и связанных с этой средой программ. GObject Introspection добавляет в GObject информацию, которая позволяет представлять созданные механизмом объекты в разных языках программирования. Если нужно создать конвейеры GStreamer с помощью Python или управлять инфраструктурой UPnP с использованием Javascript и GUPnP, GObject introspection обеспечивает единственный способ сделать это.

В этом разделе описаны средства YP для генерации и упаковки данных GObject introspection, которые представляют собой описание API, обеспечиваемого библиотеками, собранными на основе инфраструктуры GLib, в частности, механизм GObject. Файлы репозитория GObject Introspection (GIR) помещаются в пакеты -dev, файлы typelib - в основные пакеты, поскольку они упаковываются вместе с библиотеками, которые подвергаются самоанализу.

Данные создаются при сборке библиотеки путём компоновки библиотеки с небольшим исполняемым модулем, запрашивающим у библиотеки её описание, а затем выполняющим обработку этого описания. Генерация таких данных в среде кросс-компиляции осложняется тем, что библиотека предназначена для целевой архитектуры, а код должен выполняться на хосте сборки. Эта проблема решается в системе сборки OE запуском кода в QEMU. К сожалению, QEMU не всегда работает хорошо.

3.14.1. Генерация данных самоанализа

Процесс включения генерации данных самоанализа (файлы GIR) для пакета библиотеки включает несколько шагов.

1. Наследование класса gobject-introspection.
2. Проверка отсутствия блокировки самоанализа в задании и включаемых файлах. Следует проверить также отсутствие gobject-introspection-data в переменной DISTRO_FEATURES_BACKFILL_CONSIDERED и qemu-usermode в переменной MACHINE_FEATURES_BACKFILL_CONSIDERED, отключающих самоанализ.
3. Сборка образа. При возникновении ошибок, похожих на невозможность найти библиотеки .so следует найти эти библиотеки в дереве кода и добавить в задание строку вида GIR_EXTRA_LIBS_PATH = "${B}/something/.libs". Примеры использования переменной можно найти в репозитории oe-core.

4. Другие ошибки могут быть связаны с не совсем стандартной поддержкой самоанализа в пакете, приводящим к проблемам кросс-компиляции. Такие вопросы часто обсуждаются в почтовых конференциях YP.

3.14.2. Запрет генерации данных самоанализа

Если не нужно создавать данные самоанализа (например, QEMU не работает для комбинации сборочного хоста и целевой архитектуры), можно отключить создание файлов GIR любым из приведённых ниже способов.

- Добавить в конфигурацию дистрибутива строку DISTRO_FEATURES_BACKFILL_CONSIDERED = "gobject-introspection-data", которая отключит генерацию данных самоанализа с помощью QEMU, но сохранит сборку инструментов и библиотек для самоанализа (для этого не требуется QEMU).
- Добавить в конфигурацию машины строку MACHINE_FEATURES_BACKFILL_CONSIDERED = "qemu-usermode", которая отключит использование QEMU при сборке пакетов для машины. В настоящее время этот вариант применяется только заданиями с самоанализом и даёт такой же эффект, как предыдущий вариант. В будущих выпусках YP этот вариант может влиять и на другие свойства.

При отключении генерации данных самоанализа их можно получить иными способами, например, копированием из подходящего каталога sysroot или созданием на целевой платформе. Система сборки OE в настоящее время не включает поддержки таких методов.

3.14.3. Тестирование данных самоанализа для образа

Ниже описана процедура, позволяющая проверить работоспособность данных самоанализа в образе.

1. Проверка отсутствия gobject-introspection-data в переменной DISTRO_FEATURES_BACKFILL_CONSIDERED и qemu-usermode в переменной MACHINE_FEATURES_BACKFILL_CONSIDERED.
2. Сборка образа core-image-sato.
3. Запуск терминальной сессии и Python в этой сессии.
4. Ввод на терминале команд

```
>>> from gi.repository import GLib
>>> GLib.get_host_name()
```

5. Более полное описание доступно по ссылке http://python-gtk-3-tutorial.readthedocs.org/en/latest/introduction.html.

3.14.4. Известные проблемы

С поддержкой GObject Introspection связан ряд проблем, указанных ниже.

- «Падение» qemu-ppc64, не позволяющее собрать данные самоанализа для архитектуры.
- Отсутствие поддержки x32 в QEMU и связанное с этим отключение данных самоанализа.
- «Падение» временных библиотек GLib в musl и связанное с этим отключение данных самоанализа.
- Отключение данных самоанализа для некоторых пакетов в результате неспособности QEMU корректно выполнять некоторые двоичные файлы для определённой архитектуры (например, gcr, libsecret, webkit).
- Некорректная работа QEMU в пользовательском режиме при запуске 64-битовых программ на 32-битовом хосте сборки. В частности, qemumips64 не работает с i686.

3.15. Использование внешних инструментов

Для использования в процесс разработки внешних инструментов следует выполнить ряд действий.

- Разобраться с местоположением инструментов. Для их сборки потребуются дополнительные действия.
- Добавить уровень с инструментами в файл bblayers.conf через переменную BBLAYERS.
- Указать в переменной EXTERNAL_TOOLCHAIN файла local.conf местоположение инструментов.

Хорошим примером использования внешних инструментов в YP является Mentor Graphics® Sourcery G++. Информация об использовании инструментов приведена в файле README на странице http://github.com/MentorEmbedded/meta-sourcery/. Дополнительная информация приведена также в описании переменной TCMODE [3].

3.16. Создание образов с дисковыми разделами с помощью Wic

Создание образа для определённой аппаратной платформы с использованием системы сборки не обязательно ведёт к возможности загрузки этого образа на устройстве. Физические устройства принимают и загружают образы разными способами с учётом своих возможностей. Обычно информация об устройстве может подсказать нужный формат. Если для устройства нужны несколько разделов на SD-карте, флэш-диске или HDD, можно использовать OE Image Creator (Wic) для создания образа с разделами.

Команда wic создаёт образы с разделами из результатов сборки OE. Генерация образа управляется командами разбиения из файла OE kickstart (.wks), заданного в команде напрямую или выбранного из стандартных файлов, которые можно увидеть по команде wic list images (3.16.5. Использование имеющихся файлов Kickstart). Использование команды для имеющихся результатов сборки приводит к созданию одного или нескольких образов, которые можно записать на носитель и поместить в устройство. Описание файлов kickstart приведено в разделе OpenEmbedded Kickstart (.wks) Reference [3].

Команда wic и инфраструктура, на которой она основана, по определению неполны. Задача команды — разрешить создание настраиваемых образов, поэтому команда проектировалась для расширения через интерфейс подключаемых модулей, описанных в параграфе 3.16.6. Использование интрефейса подключаемых модулей Wic.

3.16.1. Основы

Здесь приведены некоторые сведения об утилите Wic, которые могут быть интересны, хотя и не требуются для работы.

- Имя Wic образовано от oeic[1], где дифтонг oe произносится как w, поскольку oeic сложней запомнить и сказать.
- Wic базируется на платформе Meego Image Creator (mic), но реализация Wic существенно переработана для прямого использования результатов сборки OE вместо установки и настройки пакетов, которые уже включены в результаты OE.
- Wic является независимой автономной утилитой, которая является более простой и гибкой в сравнении с классом OE-Core image-live. Функциональность Wic реализована базовым языком разбиения, основанным на синтаксисе Redhat kickstart.

3.16.2. Требования

Для использования Wic в системе сборки OE нужно выполнить приведённые ниже требования.

- Дистрибутив Linux на хосте сборки должен поддерживать YP (см. раздел Supported Linux Distributions [3]).
- На хосте сборки должны быть установлены стандартные системные утилиты, такие как cp.
- Следует выполнить сценарий инициализации среды (oe-init-build-env) из каталога сборки.
- Результаты сборки должны быть доступны, что обычно означает наличие собранного с помощью OE образа (например, core-image-minimal). Создание образа для генерации финального образа с помощью Wic может показаться избыточным, но для текущей версии Wic нужны результаты работы системы сборки OE.
- Нужно собрать несколько естественных инструментов для работы на хосте сборки с помощью команды bitbake parted-native dosfstools-native mtools-native.
- Нужно включить wic в переменную IMAGE_FSTYPES.
- Нужно включить имя файла kickstart в переменную WKS_FILE.

3.16.3. Доступ к справочной информации

Можно получить справку о работе с командой с помощью команды wic с опцией wic -h, wic --help или wic help. В настоящее время Wic поддерживает 7 команд - cp, create, help, list, ls, rm, write. Для получения справки по работе с ними служат команды вида wic help command. Например, команда wic help write выведет справку о работе с командой write. Wic поддерживает 3 темы справок - overview, plugins и kickstart. Можно получить справку по любой из тем командой вида wic help topic. Например, для получения обзорной справки служит команды wic help overview.

Имеется дополнительный вид справок для Wic, показывающий список доступных образов Wic.

```
$ wic list images
  mpc8315e-rdb                 Create SD card image for MPC8315E-RDB
  genericx86                   Create an EFI disk image for genericx86*
  beaglebone-yocto             Create SD card image for Beaglebone
  edgerouter                   Create SD card image for Edgerouter
  qemux86-directdisk           Create a qemu machine 'pcbios' direct disk image
  directdisk-gpt               Create a 'pcbios' direct disk image
  mkefidisk                    Create an EFI disk image
  directdisk                   Create a 'pcbios' direct disk image
  systemd-bootdisk             Create an EFI disk image with systemd-boot
  mkhybridiso                  Create a hybrid ISO image
  sdimage-bootpart             Create SD card image with a boot partition
  directdisk-multi-rootfs      Create multi rootfs image using rootfs plugin
  directdisk-bootloader-config Create a 'pcbios' direct disk image with custom bootloader config
```

Зная список образов, можно получить справку по конкретному образу с помощью команды вида

```
$ wic list beaglebone-yocto help
Creates a partitioned SD card image for Beaglebone.
Boot files are located in the first vfat partition.
```

3.16.4. Режимы работы

В зависимости от уровня управления результатами вывода системы сборки OE доступны 2 режима - Raw и Cooked:

- *режим Raw* позволяет явно указать результаты сборки параметрами команды Wic;
- *режим Cooked* использует текущую установку MACHINE и имя образа для автоматического выбора результатов сборки, включаемых в образ; нужно лишь указать файл kickstart и имя образа для включения результатов сборки.

В любом случае результаты сборки должны быть доступны в момент ввода команды.

3.16.4.1. Режим Raw

Запуск Wic в режиме raw позволяет указать все разделы в строке команды. Основным применением этого режима является создание образов при сборке ядра за пределами дерева YP. Иными словами, можно указать произвольное ядро, размещение корневой файловой системы и т. п. В режиме cooked просматривается лишь каталог сборки (например, tmp/deploy/images/machine). Команда wic имеет вид wic create wks_file options ...

wks_file

Файл OpenEmbedded kickstart. Можно использовать свой файл или один из имеющихся в системе сборки.
Дополнительные аргументы рассмотрены ниже.

[1]OpenEmbedded Image Creator - генератор образов OpenEmbedded.

-h, --helpshow
выводит справочную информацию и завершает работу.
-o OUTDIR, --outdir OUTDIR
каталог для создания образа.
-e IMAGE_NAME, --image-name IMAGE_NAME
имя образа для использования результатов сборки (например, core-image-sato).
-r ROOTFS_DIR, --rootfs-dir ROOTFS_DIR
путь к каталогу /rootfs для использования в качестве источника .wks rootfs.
-b BOOTIMG_DIR, --bootimg-dir BOOTIMG_DIR
путь к каталогу с элементами загрузки (например, /EFI или /syslinux) для использования в .wks bootimg.
-k KERNEL_DIR, --kernel-dir KERNEL_DIR
путь к каталогу с ядром для использования в .wks bootimg
-n NATIVE_SYSROOT, --native-sysroot NATIVE_SYSROOT
путь к native sysroot с инструментами, использованными при сборке образа.
-s, --skip-build-check
задаёт пропуск проверки сборки.
-f, --build-rootfs
rootfs для сборки.
-c {gzip,bzip2,xz}, --compress-with {gzip,bzip2,xz}
задаёт алгоритм сжатия образа.
-m, --bmapgenerate
.bmap
--no-fstab-update
отключает изменения файла fstab.
-v VARS_DIR, --vars VARS_DIR
каталог с файлами <image>.env. Содержащими переменные bitbake.
-D, --debug
задаёт вывод отладочной информации.
Для запуска Wic не нужны полномочия root и даже не следует использовать утилиту от имени root.

3.16.4.2. Режим Cooked

Wic в режиме cooked использует элементы из каталога сборки. Иными словами, не нужно указывать размещение ядра и корневой файловой системы, достаточно указать файл kickstart и образ из которого берутся результаты сборки, с помощью опции -e. Wic в режиме Cooked запускается командой вида wic create wks_file -e IMAGE_NAME.

wks_file
Имя файла OE kickstart. Можно использовать свой файл или один из файлов в составе выпуска YP.
Обязательный аргумент
-e IMAGE_NAME, --image-name IMAGE_NAME - имя образа для использования результатов сборки (например, core-image-sato)

3.16.5. Использование имеющихся файлов Kickstart

Можно не создавать свой файл kickstart, воспользовавшись готовыми из состава Wic. Эти файлы можно найти в репозитории YP (каталог poky/meta-yocto-bsp/wic или poky/scripts/lib/wic/canned-wks). Для просмотра доступных файлов можно использовать команду wic list images.

```
$ wic list images
  mpc8315e-rdb                 Create SD card image for MPC8315E-RDB
  genericx86                   Create an EFI disk image for genericx86*
  beaglebone-yocto             Create SD card image for Beaglebone
  edgerouter                   Create SD card image for Edgerouter
  qemux86-directdisk           Create a qemu machine 'pcbios' direct disk image
  directdisk-gpt               Create a 'pcbios' direct disk image
  mkefidisk                    Create an EFI disk image
  directdisk                   Create a 'pcbios' direct disk image
  systemd-bootdisk             Create an EFI disk image with systemd-boot
  mkhybridiso                  Create a hybrid ISO image
  sdimage-bootpart             Create SD card image with a boot partition
  directdisk-multi-rootfs      Create multi rootfs image using rootfs plugin
  directdisk-bootloader-config Create a 'pcbios' direct disk image with custom bootloader config
```

При использовании имеющегося файла не нужно указывать расширение .wks. Ниже приведён пример использования файла directdisk в режиме Raw.

```
$ wic create directdisk -r rootfs_dir -b bootimg_dir -k kernel_dir -n native_sysroot
```

Фактические команды языка разбиения, используемые в файле genericx86.wks для создания образа, показаны ниже.

```
# short-description: Create an EFI disk image for genericx86*
# long-description: Creates a partitioned EFI disk image for genericx86* machines
part /boot --source bootimg-efi --sourceparams="loader=grub-efi" --ondisk sda --label msdos --active --align 1024
part / --source rootfs --ondisk sda --fstype=ext4 --label platform --align 1024 --use-uuid
part swap --ondisk sda --size 44 --label swap1 --fstype=swap

bootloader --ptable gpt --timeout=5 --append="rootfstype=ext4 console=ttyS0,115200 console=tty0"
```

3.16.6. Использование интерфейса подключаемых модулей Wic

Можно расширить и настроить Wic с помощью подключаемых модулей (plug-in). Плагины Wic состоят из двух частей - source и imager (не рассматриваются здесь). Модули source обеспечивают механизм настройки содержимого разделов в процессе генерации образа. Можно использовать модули source для отображения значений, указываемых командами --source в файлах *.wks на реализацию модуля, применяемого для заполнения раздела.

При использовании модулей, связанных зависимостями (например, от инструментов native, загрузчиков и т. п.) во время построения образа Wic, нужно указать эти зависимости в переменной WKS_FILE_DEPENDS. Модули source

являются субклассом, определяемым в файлах plug-in. Некоторые из таких файлов входят в состав YP. Каждый файл содержит плагины source, предназначенные для заполнения разделов в образах Wic. Класс SourcePlugin, к которому относятся модули source, определён в файле poky/scripts/lib/wic/pluginbase.py.

Можно реализовать модули source на уровне, лежащем вне репозитория исходных кодов (внешний уровень). При этом они должны находиться в каталоге scripts/lib/wic/plugins/source/ внутри этого уровня, чтобы быть доступными для Wic.

Когда реализации Wic нужно вызвать зависящую от раздела реализацию, просматриваются плагины с именем, указанным параметром --source в файле kickstart для этого раздела. Например, при установке раздела с помощью команды part /boot --source bootimg-pcbios --ondisk sda --label boot --active --align 1024 используются методы, определённые как члены класса соответствующего подключаемого модуля source (bootimg-pcbios) в файле bootimg-pcbios.py.

Ниже приведено соответствующее определение плагина из the bootimg-pcbios.py для предыдущей команды вместе с примером метода, вызываемого реализацией Wic для подготовки раздела с зависимой от реализации функцией.

```
      ...
     class BootimgPcbiosPlugin(SourcePlugin):
         """
         Create MBR boot partition and install syslinux on it.
         """

         name = 'bootimg-pcbios'
      ...
         @classmethod
         def do_prepare_partition(cls, part, source_params, creator, cr_workdir,
          oe_builddir, bootimg_dir, kernel_dir,
          rootfs_dir, native_sysroot):
 """
 Called to do the actual content population for a partition i.e. it
 'prepares' the partition to be incorporated into the image.
 In this case, prepare content for legacy bios boot partition.
 """
      ...
```

Если субкласс (plug-in) сам не реализует определённую функцию, Wic находит и использует принятую по умолчанию версию надкласса (superclass). По этой причине все модули source выводятся из класса SourcePlugin. Этот класс, определённый в pluginbase.py, включает набор методов, которые модули source plug-ins могут использовать или переопределять. Все плагины (субкласс SourcePlugin), не реализующие тот или иной метод, наследуют его реализацию от класса SourcePlugin. Описание SourcePlugin можно найти в файле pluginbase.py, а ниже приведён список реализованных в этом классе методов.

- *do_prepare_partition()* вызывается для заполнения раздела содержимым. Иными словами, метод готовит окончательный образ раздела для встраивания в образ диска.
- *do_configure_partition()* вызывается перед do_prepare_partition() для создания конфигурационных файлов раздела (например, для syslinux или grub).
- *do_install_disk()* вызывается после того, как все разделы подготовлены и собраны в образ диска. Этот метод обеспечивает завершение записи образа диска (например, запись MBR).
- *do_stage_partition()* - специальный метод подготовки содержимого перед вызовом do_prepare_partition(). Обычно этот метод пуст. Раздел, как правило, просто использует переданные параметры (например, неизменное значение bootimg_dir), однако в некоторых случаях может потребоваться особый подход. Например, могут потребоваться некоторые файлы из bootimg_dir + /boot. Этот метод позволяет размещать такие файлы индивидуально. Метод get_bitbake_var() обеспечивает доступ к нестандартным переменным, которые могут применяться в таких случаях.

Механизм модулей source можно расширить путём добавления «ловушек», создания дополнительных методов в SourcePlugin и соответствующих производных субклассов. Код, вызывающий методы, использует функцию plugin.get_source_plugin_methods() для поиска требуемых методов. Поиск реализуется с помощью ключей, содержащих имена методов.

3.16.7. Примеры

В этом разделе приведено несколько примеров использования утилиты Wic. Предполагается, что выполнены все требования раздела 3.16.2. Требования и применяется заранее созданный образ core-image-minimal.

3.16.7.1. Создание образа с имеющимся файлом Kickstart

Здесь используется режим Cooked с kickstart-файлом mkefidisk.

```
$ wic create mkefidisk -e core-image-minimal
INFO: Building wic-tools...
   .
   .
   .
INFO: The new image(s) can be found here:
  ./mkefidisk-201804191017-sda.direct

The following build artifacts were used to create the image(s):
ROOTFS_DIR: /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0/rootfs
BOOTIMG_DIR:   /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0/recipe-sysroot/
usr/share
KERNEL_DIR: /home/stephano/build/master/build/tmp-glibc/deploy/images/qemux86
NATIVE_SYSROOT: /home/stephano/build/master/build/tmp-glibc/work/i586-oe-linux/wic-tools/1.0-r0/recipe-sysroot-native

INFO: The image(s) were created using OE kickstart file:
       /home/stephano/build/master/openembedded-core/scripts/lib/wic/canned-wks/mkefidisk.wks
```

Это простейший способ создать образ в режиме cooked с указанием kickstart-файла и опции -e для задания имеющихся результатов сборки. В файле local.conf переменная MACHINE должна указывать используемую машину (qemux86).

Когда образ собран, выходные данные указывают местоположение образа, использованные результаты и информацию файла kickstart. Следует внимательно проверить эту информацию, чтобы убедиться в корректности собранного образа.

Полученный образ можно записать из каталога сборки на USB-носитель или иную среду и загрузить систему с неё. Для записи можно использовать команду bmaptool или dd, как показано ниже.

```
$ oe-run-native bmaptool copy mkefidisk-201804191017-sda.direct /dev/sdX
```

или

```
$ sudo dd if=mkefidisk-201804191017-sda.direct of=/dev/sdX
```

Информация об использовании bmaptool приведена в разделе 3.17. Запись образов с помощью bmaptool.

3.16.7.2. Использование изменённого файла Kickstart

Поскольку созданием образа с разделами управляет файл kickstart, изменение параметров файле позволяет влиять на получаемый образ. Ниже приведён пример этого с использованием файла directdisk-gpt.

Как было отмечено, команда wic list images выводит список имеющихся kickstart-файлов. Файл directdisk-gpt.wks размещается в каталоге scripts/lib/image/canned-wks/ внутри дерева исходных кодов (например, poky). В этом каталоге размещаются другие файлы и можно создать свой kickstart-файл. Имеющийся файл directdisk-gpt содержит почти все, что нужно для примера. Однако для загрузки образа в примере будет применяться устройство sdb вместо sda, указанного в файле directdisk-gpt.

В примере сначала создаётся копия файла directdisk-gpt.wks с другим именем в каталоге scripts/lib/image/canned-wks.

```
$ cp /home/stephano/poky/scripts/lib/wic/canned-wks/directdisk-gpt.wks \
     /home/stephano/poky/scripts/lib/wic/canned-wks/directdisksdb-gpt.wks
```

Затем ф копии directdisksdb-gpt.wks строка "--ondisk sda" заменяется на "--ondisk sdb", как показано ниже.

```
part /boot --source bootimg-pcbios --ondisk sdb --label boot --active --align 1024
part / --source rootfs --ondisk sdb --fstype=ext4 --label platform --align 1024 --use-uuid
```

Этот файл будет создавать образ directdisksdb-gpt с результатами сборки core-image-minimal для машины Next Unit of Computing (nuc), указанной переменной MACHINE в файле local.conf.

```
$ wic create directdisksdb-gpt -e core-image-minimal
INFO: Building wic-tools...
...
Initialising tasks: 100% |########################################| Time: 0:00:01
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 1161 tasks of which 1157 didn't need to be rerun and all succeeded.
INFO: Creating image(s)...

INFO: The new image(s) can be found here:
       ./directdisksdb-gpt-201710090938-sdb.direct

The following build artifacts were used to create the image(s):
ROOTFS_DIR:    /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0/
rootfs
BOOTIMG_DIR:   /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0/
recipe-sysroot/usr/share
KERNEL_DIR:        /home/stephano/build/master/build/tmp-glibc/deploy/images/qemux86
NATIVE_SYSROOT:     /home/stephano/build/master/build/tmp-glibc/work/i586-oe-linux/wic-tools/1.0-r0/recipe-
sysroot-native

INFO: The image(s) were created using OE kickstart file:
  /home/stephano/poky/scripts/lib/wic/canned-wks/directdisksdb-gpt.wks
```

В продолжение примера образ записывается с помощью команды dd на накопитель USB или иную среду для загрузки.

```
$ sudo dd if=directdisksdb-gpt-201710090938-sdb.direct of=/dev/sdb
140966+0 records in
140966+0 records out
72174592 bytes (72 MB, 69 MiB) copied, 78.0282 s, 925 kB/s
$ sudo eject /dev/sdb
```

3.16.7.3. Использование изменённого файла Kickstart и работа в режиме Raw

В следующем примере вручную задаются все включаемые в образ результаты сборки (режим Raw) и используется изменённый файл kickstart. Применяется также опция -o, заставляющая Wic создавать в отличном от принятого по умолчанию каталоге.

```
$ wic create /home/stephano/my_yocto/test.wks -o /home/stephano/testwic \
--rootfs-dir /home/stephano/build/master/build/tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/rootfs \
--bootimg-dir      /home/stephano/build/master/build/tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/recipe-sysroot/
usr/share \
--kernel-dir /home/stephano/build/master/build/tmp/deploy/images/qemux86 \
--native-sysroot /home/stephano/build/master/build/tmp/work/i586-poky-linux/wic-tools/1.0-r0/recipe-sysroot-native

INFO: Creating image(s)...

INFO: The new image(s) can be found here:
  /home/stephano/testwic/test-201710091445-sdb.direct

The following build artifacts were used to create the image(s):
ROOTFS_DIR:       /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0/rootfs
BOOTIMG_DIR:          /home/stephano/build/master/build/tmp-glibc/work/qemux86-oe-linux/core-image-minimal/1.0-r0/recipe-
sysroot/usr/share
KERNEL_DIR:       /home/stephano/build/master/build/tmp-glibc/deploy/images/qemux86
NATIVE_SYSROOT:   /home/stephano/build/master/build/tmp-glibc/work/i586-oe-linux/wic-tools/1.0-r0/recipe-sysroot-native
```

```
INFO: The image(s) were created using OE kickstart file:
  /home/stephano/my_yocto/test.wks
```

В этом примере переменная MACHINE не задана в файле local.conf, поэтому результаты сборки указываются вручную.

3.16.7.4. Использование Wic для манипуляций с образом

Возможности Wic позволяют сократить время разработки образа. Например, можно использовать Wic для удаления раздела ядра из образа, а затем поместить туда новый образ ядра. Это позволит не собирать образ полностью, ограничившись сборкой образа ядра.

В рассматриваемом ниже примере предполагается наличие на хосте разработки установленного пакета mtools. В примере из образа Wic удаляется удаляется ядро, затем в образ помещается новое ядро.

1. *Просмотр списка разделов* с помощью команды wic ls.

```
$ wic ls tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic
Num     Start        End          Size      Fstype
 1       1048576     25041919     23993344  fat16
 2      25165824     72157183     46991360  ext4
```

   Вывод показывает два раздела в образе core-image-minimal-qemux86.wic.

2. *Проверка отдельного раздела* с помощью команды wic ls.

```
$ wic ls tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic:1
Volume in drive : is boot
 Volume Serial Number is E894-1809
Directory for ::/

libcom32 c32    186500 2017-10-09  16:06
libutil  c32     24148 2017-10-09  16:06
syslinux cfg       220 2017-10-09  16:06
vesamenu c32     27104 2017-10-09  16:06
vmlinuz        6904608 2017-10-09  16:06
 5 files           7 142 580 bytes
      16 582 656 bytes free
```

   Вывод команды показывает 5 файлов и файл vmlinuz содержит ядро. Если при работе команды возникает показанная ниже ошибка, следует обновить файл ~/.mtoolsrc, убедившись в наличии там строки “mtools_skip_check=1“, и повторить команду Wic.

```
ERROR: _exec_cmd: /usr/bin/mdir -i /tmp/wic-parttfokuwra ::/ returned '1' instead of 0
output: Total number of sectors (47824) not a multiple of sectors per track (32)!
Add mtools_skip_check=1 to your .mtoolsrc file to skip this test
```

3. *Удаление имеющегося ядра (vmlinuz)* с помощью команды wic rm.

```
$ wic rm tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic:1/vmlinuz
```

4. *Добавление нового ядра* с помощью команды wic cp. Местоположение нового ядра зависит от способа его сборки. При использовании devtool и SDK ядро записывается в каталог tmp/work directory расширяемого SDK, а при использовании команды make - в каталог workspace/sources. В примере предполагается, что ядро собрано с помощью devtool.

```
wic    cp    ~/poky_sdk/tmp/work/qemux86-poky-linux/linux-yocto/4.12.12+git999-r0/linux-yocto-4.12.12+git999/arch/
x86/boot/bzImage \
~/poky/build/tmp/deploy/images/qemux86/core-image-minimal-qemux86.wic:1/vmlinuz
```

   После возврата файла ядра в образ можно использовать команду dd или bmaptool для записи образа на карту SD или носитель USB. Отметим, что bmaptool работает в 10 - 20 быстрее, нежели dd.

3.17. Запись образов с помощью bmaptool

Быстрый и простой способ записи образа на загрузочное устройство обеспечивает программа Bmaptool из состава системы сборки OE. Bmaptool является инструментом общего назначения, создающим отображение блоков (bmap) и применяющим его для копирования файла. По сравнению с традиционными средствами, такими как dd и cp, Bmaptool может копировать (записывать) большие файлы существенно быстрее.

При работе с Ubuntu или Debian можно установить пакет bmap-tools и после этого использовать инструмент без указания PATH даже от имени root. Если установить bmap-tools невозможно, потребуется собрать Bmaptool с помощью команды bitbake bmap-tools-native.

Ниже приведён пример записи образа Wic с использованием Bmaptool.

1. *Обновление файла local.conf* путём включения строки IMAGE_FSTYPES += "wic wic.bmap".
2. *Подготовка образа.* Если образе не был собран заранее с использованием указанной выше переменной IMAGE_FSTYPES, его нужно собрать с помощью команды bitbake image.
3. *Запись образа на устройство* с использованием Bmaptool зависит от конкретных настроек. Здесь предполагается, что образ хранится в каталоге deploy/images/ внутри каталога сборки.
   - При наличии прав записи в среду используется команда вида (devX нужно заменить реальным именем).

     ```
     $ oe-run-native bmap-tools-native bmaptool copy build-directory/tmp/deploy/images/machine/image.wic /dev/sdX
     ```

   - Если прав записи в среду нет, нужно сначала предоставить их с помощью команды sudo chmod 666 /dev/sdX, а затем ввести приведённую выше команду записи.

Для получения справки о работе с bmaptool служит команда bmaptool --help.

3.18. Повышение защищенности образов

Безопасность важна для встраиваемых систем. Ниже приведён ряд ссылок по этим вопросам.

- *"Security Risks of Embedded Systems",* Bruce Schneier.
- *"Internet Census 2012",* Carna Botnet.
- *"Security Issues for Embedded Devices",* Jake Edge.

Для защиты образа имеются действия, инструменты и переменные, которые помогут обеспечить безопасность целевого устройства. В этом разделе приведены рекомендации, учёт которых позволит сделать образ защищённым. Поскольку требования безопасности и риски отличаются для разных устройств, здесь не представлено полной информации по защите ОС. Настоятельно рекомендуется обратиться к другим источникам информации по защите встраиваемых систем на базе Linux и вопросам безопасности.

3.18.1. Общие вопросы

Ниже перечислены некоторые предложения общего плана, учёт которых позволит защитить устройство.

- Проверка кода, добавляемого в систему (например, приложений), с помощью инструментов статического анализа позволит обнаружить переполнение буферов и другие возможные уязвимости.
- Особое внимание к защите административных web-интерфейсов, которые зачастую работают с повышенными привилегиями. Это ведёт к росту рисков в результате нарушения защиты таких интерфейсов. Следует обратить внимание на основные уязвимости web-систем, такие как кросс-сайтовые сценарии (XSS), непроверяемый ввод и т. п. Как и для системных паролей учётные данные для доступа к web-интерфейсу не должны быть одинаковыми на всех системах. Это особенно важно в тех случаях, когда интерфейс включён по умолчанию, поскольку не все пользователи поменяют сразу учётные записи.
- Возможность обновления программ на устройстве для устранения обнаруженных уязвимостей, что особо важно для устройств с доступом в сеть.
- Удаление или запрет функций отладки в окончательном образ, как описано в параграфе 3.18.3. Вопросы, связанные с системой сборки OE.
- Отключение или удаление ненужных сетевых служб.
- Удаление из образа ненужных программ.
- Включение аппаратной поддержки защищённой загрузки, если устройство имеет её.

3.18.2. Флаги защиты

YP поддерживает флаги защиты, которые позволяют повысить уровень безопасности образов. Флаги защиты определены в файле meta/conf/distro/include/security_flags.inc дерева источников (например, poky). В зависимости от заданий некоторые флаги могут быть установлены или сброшены по умолчанию. Для использования флагов защиты следует включить в файл local.conf или конфигурацию дистрибутива строку require conf/distro/include/security_flags.inc.

3.18.3. Вопросы, связанные с системой сборки OE

Ниже перечислены некоторые действия по повышению уровня защиты образов, связанные с системой сборки OE.

- Следует исключить debug-tweaks из выбранных в IMAGE_FEATURES свойств. При создании нового проекта в local.conf по умолчанию включается строка EXTRA_IMAGE_FEATURES = "debug-tweaks", для отключения которой достаточно поместить в начало строки символ комментария или исключить из строки debug-tweaks. Эта установка делает пустым пароль root, что может быть полезно при отладке в процессе разработки.
- Можно установить пароль root для образа, а также пароли иных пользователей (например, администраторов). При установке не следует задавать один пароль для разных образов или пользователей. Предпочтительным методом установки паролей является наследование класса extrausers (см. раздел extrausers.bbclass [3]).
- Следует рассмотреть вопрос включения инфраструктуры обязательного контроля доступа (MAC), такой как SMACK или SELinux и её соответствующей настройки на устройстве (см. уровень meta-selinux).

3.18.4. Средства усиления защиты образа

YP включает средства защиты создаваемых образов, которые можно найти на уровне meta-security в Yocto Project Source Repositories.

3.19. Создание своего дистрибутива

При сборке образа с помощью YP без изменения метаданных дистрибутива будет создан дистрибутив Poky. Для более эффективного управления выбором приложений, опциями компиляции и настройками на нижних уровнях можно создать свой дистрибутив. Процесс включает создание уровня дистрибутива, конфигурационного файла, а также добавления нужного кода и уровня метаданных, как описано ниже.

- *Создание уровня для нового дистрибутива,* позволяющего хранить метаданные и код. Использование отдельного уровня упрощает воспроизведение конфигурации сборки для разных машин. Создание уровней общего назначения описано в параграфе 3.1.8. Создание базового уровня с помощью сценария bitbake-layers.
- *Создание файла конфигурации дистрибутива* в каталоге conf/distro на уровне дистрибутива. Файл следует назвать в соответствии с именем дистрибутива (например, mydistro.conf). Имя дистрибутива указывается в переменной DISTRO файла local.conf.

  Можно выделить часть конфигурации во включаемые файлы и затем «требовать» (require) их в файле конфигурации дистрибутива. Включаемые файлы следует размещать в каталоге conf/distro/include. Типичным примером использования включаемых файлов является разделения заданий по номеру версии и выпуску.

В файле конфигурации должны быть заданы переменные DISTRO_NAME и DISTRO_VERSION. Переменные DISTRO_FEATURES, DISTRO_EXTRA_RDEPENDS, DISTRO_EXTRA_RRECOMMENDS, TCLIBC не обязательны, но обычно включаются в файл конфигурации дистрибутива.

Можно создать файл на основе базовой конфигурации из OE-Core (файл conf/distro/defaultsetup.conf) и просто изменить соответствующие переменные. Другим вариантом является создание файла конфигурации с нуля с использованием defaultsetup.conf или файла конфигурации из другого дистрибутива в качестве образца.

- *Включение переменных,* которые задают принятые по умолчанию значения или устанавливают значения как часть конфигурации дистрибутива. Можно включать почти все переменные из файла local.conf.
- *Указание файла конфигурации дистрибутива* в файле local.conf каталога сборки через переменную DISTRO. Например, если файл конфигурации называется mydistro.conf, указывается строка DISTRO = "mydistro".
- *Добавление других компонент уровня.*
  - Добавление заданий для установки специфических для дистрибутива файлов конфигурации, которые не включены в другие задания. Если такие файлы уже имеются в других заданиях, следует создать для них файлы дополнения (.bbappend). Общее описание и рекомендации по добавлению заданий на уровень приведены в параграфах 3.1.1. Создание своего уровня и 3.1.2. Рекомендации по организации уровней.
  - Добавление заданий, относящихся к дистрибутиву.
  - Включение файла дополнения psplash для фирменной заставки, как описано в параграфе 3.1.5. Использование файлов .bbappend на уровне.
  - Включение других файлов дополнения, вносящих изменения в имеющиеся задания.

3.20. Создание шаблона каталога конфигурации

При распространении своей системы сборки другим пользователям может возникнуть желание изменить сообщение, выводимое сценарием установки, а изменить шаблоны файлов конфигурации (local.conf и bblayers.conf) в каталоге сборки.

Система сборки OE использует переменную TEMPLATECONF для указания каталога с данными конфигурации, которые в конечном итоге попадают в каталог conf внутри каталога сборки. По умолчанию задано TEMPLATECONF=${TEMPLATECONF:-meta-poky/conf}. Этот каталог используется системой сборки для хранения шаблонов, из которых создаются некоторые файлы конфигурации. В частности, там содержатся файлы bblayers.conf.sample, local.conf.sample и conf-notes.txt, из которых система сборки создаёт файлы bblayers.conf и local.conf, а также список целей BitBake.

Для переопределения принятых по умолчанию конфигураций в каждом новом каталоге сборки нужно лишь указать каталог в TEMPLATECONF в файле .templateconf на верхнем уровне дерева источников (например, poky).

Практика рекомендует помещать каталог шаблонов конфигурации на уровень дистрибутива. Например, для уровня meta-mylayer в домашнем каталоге можно создать каталог шаблонов конфигурации myconf. Описанное изменение .templateconf заставит систему сборки OE просматривать каталог и создавать файлы конфигурации на основе найденных файлов *.sample. Окончательные файлы конфигурации (local.conf и bblayers.conf) в конечно итоге будут помещены в каталог сборки, но созданы будут на основе файлов *.sample.

```
TEMPLATECONF=${TEMPLATECONF:-meta-mylayer/myconf}
```

Кроме файлов *.sample в принятом по умолчанию каталоге meta-poky/conf хранится также файл conf-notes.txt. Сценарий организации среды сборки (oe-init-build-env) использует этот файл для вывода целей BitBake в процессе выполнения сценария. Редактирование conf-notes.txt обеспечивает возможность указать нужные цели. Ниже показан список целей, выводимых в результате работы сценария.

```
You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support
```

3.21. Экономия дискового пространства при сборке

Для экономии дискового пространства при сборке можно добавить в файл local.conf каталога сборки строку INHERIT += "rm_work" для удаления рабочего каталога задания после сборки этого задания (см. rm_work [3]).

3.22. Работа с пакетами

3.22.1. Исключение пакетов из образа

Иногда возникает потребность исключить некоторые пакеты из образа, для чего имеется несколько переменных, указывающих системе сборки необходимость игнорировать рекомендуемые пакеты или совсем не устанавливать пакет. Каждая из перечисленных ниже переменных работает только с пакетами IPK и RPM, но не работает с Debian. Эти переменные можно указывать в файле local.conf file или привязывать к заданию для конкретного образа путём переопределения имени задания. Более подробная информация о переменных представлена в [3].

- BAD_RECOMMENDATIONS указывает пакеты, которые не следует устанавливать.
- NO_RECOMMENDATIONS отменяет установку всех пакетов recommended-only.
- PACKAGE_EXCLUDE исключает пакет из образа независимо от установки recommended-only. Следует помнить, что это может приводить к отказу, если пакет требуется для установки другого пакета.

3.22.2. Инкрементирование версии пакета

Здесь рассматривается управление нумерацией версий двоичных пакетов и рассмотрены некоторые переменные и термины. Для понимания нумерации версий пакетов нужно принимать во внимание ряд аспектов, указанных ниже.

- *Двоичный пакет* - это то, что в конечном итоге устанавливается в образ.
- *Версия двоичного пакета* состоит из двух компонент - версия и выпуск (revision).
- *Эпоха* (PE) также рассматривается, но в большинстве случаев PE игнорируется.
- *Версия и выпуск* берутся из переменных PV и PR.
- *PV* указывает версию задания и представляет версию программы. Не следует путать PV с версией двоичного пакета.
- *PR* - выпуск задания.
- *SRCPV* - строка, используемая системой сборки OE для помощи в определении значения PV, когда нужно включить выпуск исходного кода.
- *PR Service* - сетевая служба, помогающая автоматизировать запись пакетов в хранилища, совместимые с имеющимися менеджерами пакетов, такими как RPM, APT, OPKG.

При каждом изменении содержимого двоичного пакета номер его версии должен меняться. Это обеспечивается путём изменения (bumping) значений PR и/или PV, которое может выполняться одним из двух способов:

- автоматически с использованием службы выпусков пакетов (PR Service);
- вручную путём инкрементирования значений переменных PR и/или PV.

С учётом того, что основной задачей системы сборки и её пользователей является поддержка хранилища пакетов, совместимого с имеющимися менеджерами пакетов, такими как RPM, APT и OPKG, предпочтительно использование автоматизированной системы управления версиями. В любой системе основным требованием к версиям двоичных пакетов является монотонное возрастание и наличие компонент версии, поддерживающих это. Поддержка монотонного роста версий описана в параграфе 3.22.2.3. Автоматическое инкрементирование номера версии пакета.

3.22.2.1. Работа с сервисом PR

Как уже отмечалось, попытка сохранять номера выпусков в метаданных подвержена ошибкам, неточна и создаёт проблемы для людей, представляющих задания. Служба PR автоматически создаёт возрастающие номера, в частности для выпуска. Информация о службе PR доступна на странице PR Service.

В YP для упрощения нумерации версий используются переменные PE, PV и PR для эпохи, версии и выпуска. Значения переменных зависят от правил и процедур дистрибутива и хранилища пакетов.

Поскольку система сборки OE использует подписи, которые уникальны для данной сборки, она знает, когда нужно заново собрать пакет. Весь ввод в данную задачу представляется подписью и её изменение может служить сигналом для повторной сборки. Таким образом, система сборки не использует переменные PR, PV, PE для запуска повторной сборки, однако для создания значений этих переменных могут применяться подписи.

Служба PR работает с генераторами OEBasic и OEBasicHash. Значение PR увеличивается при изменении контрольной суммы, а эти изменения вызываются механизмами генератора. Система сборки включает значения из службы PR в поле PR как дополнение, используя форму .x, так что r0 становится r0.1, r0.2 и т. д. Эта схема позволяет использовать имеющиеся значения PR в всех случаях, включая увеличение PR вручную, когда это требуется.

По умолчанию служба PR выключена и не запущена. В результате пакеты генерируются как «самосогласованные». Система сборки добавляет и удаляет пакеты и нет никаких гарантий по части обновления, но образы будут соответствовать внесённым изменениям. Простейшей формой службы PR является её организация на хосте сборки, где создаются пакеты для хранилища. В этом случае локальную службу PR можно включить, указав в файле local.conf внутри каталога сборки значение PRSERV_HOST = "localhost:0". После запуска службы пакеты будут автоматически получать возрастающие значения PR и BitBake будет обеспечивать запуск и остановку службы.

При наличии нескольких хостов разработки, работающих с одним хранилищем пакетов, будет использоваться общая служба PR для всех систем сборки. В этом случае службу PR нужно запускать командой bitbake-prserv --host ip --port port --start. Кроме того, нужно обновить файл local.conf на каждом хосте сборки, как описано выше, указав сервер и порт.

Рекомендуется также применять историю сборки, которая добавляет некоторые проверки корректности версий двоичных пакетов. Для этого на каждом хосте сборки нужно добавить в файл local.conf приведённые ниже строки.

```
# It is recommended to activate "buildhistory" for testing the PR service
INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "1"
```

История сборки описана в разделе 3.28. Поддержка качества вывода сборки.

Система сборки OE не поддерживает данные PR как часть общего состояния (sstate). Если sstate поддерживается, все системы сборки должны использовать общую службу PR или эта служба должна быть отключена на всех хостах сборки. Информация о sstate приведена в разделе Shared State Cache [1].

3.22.2.2. Увеличение PR вручную

Вместо установки службы PR можно вручную увеличивать переменную PR. Если зафиксированные изменения приводят в изменению вывода пакета, значение переменной PR должно быть увеличено при фиксации изменений. Для новых заданий следует устанавливать принятое по умолчанию значение r0. Хотя значение r0 задано по умолчанию, его добавление в новое задание снижает вероятность забыть об увеличении переменной при обновлении задания.

При использовании файла .inc для нескольких заданий можно использовать также переменную INC_PR, чтобы все использующие этот файл задания собирались заново при изменении файла .inc. В этом файле должна устанавливаться переменная INC_PR (исходно r0), а в заданиях нужно установить изначально PR = "${INC_PR}.0", увеличивая последнюю часть при обновлении задания. При обновлении файла .inc увеличивается INC_PR.

При обновлении версии двоичного пакета, предполагающем изменение PV, переменную PR следует сбрасывать в r0 (или ${INC_PR}.0 при использовании INC_PR). Обычно увеличение версии происходит лишь для двоичных пакетов. Однако при изменении PV без увеличения можно увеличить переменную PE (эпоха пакета), для которой по умолчанию установлено значение 0.

Нумерацию версий пакетов рекомендуется вести в соответствии с Debian Version Field Policy Guidelines, где определён механизм сравнения версий и понятие увеличения.

3.22.2.3. Автоматическое инкрементирование номера версии пакета

При выборке из репозитория BitBake использует переменную SRCREV для определения конкретного выпуска кода, который будет служить для сборки. Можно установить SRCREV = "${AUTOREV}", чтобы система сборки OE автоматически применяла последний выпуск программы. Кроме того, нужно указать SRCPV в переменной PV для автоматического обновления версии при изменении выпуска исходного кода, например, PV = "1.0+git${SRCPV}". Система сборки OE будет подставлять вместо SRCPV значение AUTOINC+source_code_revision, заменяя AUTOINC числом в зависимости от состояния службы PR.

- Если служба PR включена, система сборки инкрементирует номер, подобно поведению PR. Это ведёт к монотонному росту номера версии пакета, например,

```
hello-world-git_0.0+git0+b6558dd387-r0.0_armv7a-neon.ipk
hello-world-git_0.0+git1+dd2f5c3565-r0.0_armv7a-neon.ipk
```

- Если служба PR выключена, система сборки заменяет AUTOINC нулём (0). Это ведёт к смене версии пакета, поскольку включается выпуск программы, но изменение номера версии не будет монотонным, например,

```
hello-world-git_0.0+git0+b6558dd387-r0.0_armv7a-neon.ipk
hello-world-git_0.0+git0+dd2f5c3565-r0.0_armv7a-neon.ipk
```

3.22.3. Обработка пакетов с необязательными модулями

Многие программы включают необязательные модули (plug-in), сборка которых зависит от опций конфигурации. Чтобы избежать дублирования логики включения модулей в задание и необходимости указывать модули вручную, система сборки OE поддерживает функции динамической упаковки модулей, для чего нужно:

- обеспечить действительную сборку модулей;
- обеспечить выполнение заданием всех зависимостей модулей от других заданий.

3.22.3.1. Обеспечение сборки модулей

Обеспечить реальное создание пакетов с модулями можно с помощью функции do_split_packages внутри функции Python populate_packages в задании. Функция отыскивает шаблоны файлов и каталогов по указанному пути и создаёт пакет для найденного, добавляя его в переменную PACKAGES и устанавливая нужные значения FILES_packagename, RDEPENDS_packagename, DESCRIPTION_packagename и т. д. Например, для задания lighttpd

```
    python populate_packages_prepend () {
        lighttpd_libdir = d.expand('${libdir}')
        do_split_packages(d, lighttpd_libdir, '^mod_(.*)\.so$',
'lighttpd-module-%s', 'Lighttpd module for %s',
 extra_depends='')
    }
```

В примере заданно множество параметров при вызове do_split_packages:

- каталог в котором do_install ищет файлы, устанавливаемые заданием;
- регулярное выражение для сопоставления с файлами модулей в каталоге поиска; в примере следует обратить внимание на () для указания части выражения, из которой выводится имя модуля;
- шаблон для имён. пакетов;
- описание каждого пакета;
- пустая строка для extra_depends отключает заданные по умолчанию зависимости основного пакета lighttpd, поэтому при нахождении файла в ${libdir} с именем mod_alias.so создаётся пакет для него, а в переменной DESCRIPTION устанавливается значение "Lighttpd module for alias".

Для более сложных случаев имеются другие опции do_split_packages. При необходимости можно расширить логику путём задания функции-ловушки, вызываемой для каждого пакета. Можно вызывать do_split_packages несколько раз если для пакета имеется не один набор модулей. Примеры использования do_split_packages приведены в файле connman.inc file каталога meta/recipes-connectivity/connman/ в репозитории poky, а также в meta/classes/kernel.bbclass.

Ниже перечислены обязательные и дополнительные аргументы do_split_packages.

Обязательные аргументы

root
Путь поиска.

file_regex
Регулярное выражения для сопоставления файлов. Скобки () служат для маркировки части выражения, которую следует применять для вывода имени модуля (для подстановки вместо %s в других аргументах).

output_pattern
Шаблон для имён. пакетов (должен включать %s).

description
Описание для каждого пакета (должно включать %s).
Дополнительные аргументы

postinst
Сценарий пост-установки для всех пакетов (как строка).
recursive
True для рекурсивного поиска (принято по умолчанию).
hook
Функция ловушка, вызываемая для каждого совпадения с перечисленными ниже аргументами в заданном порядке:
f
полный путь к файлу или каталогу для сопоставления;
pkg
имя пакета;
file_regex
см. выше;
output_pattern
см. выше;
modulename
Имя модуля, полученное с использованием file_regex.
extra_depends
Дополнительные зависимости при работе (RDEPENDS), устанавливаемые для всех пакетов. Принятое по умолчанию значение None задаёт зависимость от основного пакета (${PN}). Если такая зависимость не нужна, следует указать пустую строку в этом параметре.
aux_files_pattern
Элементы, добавляемые в FILES для каждого пакета (одна строка или список строк для нескольких пакетов с обязательным включением %s).
postrm
Сценарий postrm для использования со всеми пакетами (как строка).
allow_dirs
True для сопоставления с каталогами (по умолчанию False).
prepend
True задаёт добавление пакетов в начало PACKAGES, False (принято по умолчанию) - в конец.
match_path
Сопоставление file_regex со всем относительным путём к корню, а не только с именем файла.
aux_files_pattern_verbatim
Элементы, добавляемые в FILES для каждого пакета с выведенными именами модулей вместо преобразования в пригодные для имени пакета (строка или список строк для нескольких пакетов с обязательным включением %s).
allow_links
True разрешает сопоставление символьных ссылок (по умолчанию False).
summary
Резюме для каждого пакета (должно включать %s), значения по умолчанию не задано.

3.22.3.2. Выполнение зависимостей

Второй частью обработки пакетов с необязательными модулями является выполнение зависимостей этих модулей от других заданий. Это можно обеспечить с помощью переменной PACKAGES_DYNAMIC. Например, для упомянутого выше задания lighttpd это будет иметь вид PACKAGES_DYNAMIC = "lighttpd-module-.*".

Имя в регулярном выражении может быть любым. В примере задано имя lighttpd-module-, которое будет префиксом во всех RDEPENDS и RRECOMMENDS для пакетов, имена которых начинаются с этого префикса. При использовании do_split_packages в соответствии с предыдущим параграфом значение, помещаемое в PACKAGES_DYNAMIC, должно соответствовать шаблону имени при вызове do_split_packages.

3.22.4. Управление пакетами в процессе работы

При сборке BitBake преобразует задание в один или несколько пакетов. Например, BitBake принимает задание bash и создаёт пакеты bash, bash-bashbug, bash-completion, bash-completion-dbg, bash-completion-dev, bash-completion-extra, bash-dbg и т. п. В образ включаются не все созданные пакеты.

В некоторых случаях может потребоваться обновление, добавление, удаление или запрос пакета не целевом устройстве при работе (т. е. без создания нового образа). Например,

- нужно обеспечить обновление развёрнутых устройств в «полевых» условиях (например, безопасность);
- нужен быстрый цикл разработки одного или нескольких приложений на целевом устройстве;
- нужно временно установить отладочные пакеты разных приложений для ускорения отладки;
- нужно развернуть на устройстве минимальный выбор пакетов с возможностью расширения в «поле».

Все эти случаи похожи на обычные дистрибутивы Linux, где устройства могут получать собранные пакеты с сервера инсталляции или обновления. Возможность установки пакетов в полевых условиях называют управлениям пакетами в процессе работы. Для такого управления потребуется машина, которая будет обеспечивать собранные пакеты и требуемые метаданные, а также инструменты для управления пакетами. В качестве сервера обновления проще всего применять сборочный хост, однако он может не быть сервером пакетов. Хост сборки может передавать результаты на другой сервер (например, доступный через Internet). Это удобно для рабочей среды, поскольку получение пакетов из среды сборки может сталкиваться с проблемой их перезаписи.

Простая сборка для единственного устройства создаёт не одну базу данных о пакетах, т. е. созданные при сборке пакеты разделяются по нескольким группам на основе таких критериев, как архитектура CPU, целевая плата и применяемая там библиотека C. Например, сборка для устройства qemux86 создаёт 3 базы данных - noarch, i586,

qemux86. Если нужно сообщить устройству qemux86 о всех доступных пакетах, потребуется отдельно указать каждую из этих баз, как это обычно делается в традиционных дистрибутивах Linux.

Управление пакетами при работе не является обязательным и не требуется для сборки и разработки. Однако для использования такого управления потребуется сделать некоторые дополнительные шаги, описанные ниже.

3.22.4.1. Вопросы сборки

При создании пакетов BitBake нужно знать об используемых форматах пакетов, которые указываются в переменной PACKAGE_CLASSES. Для этого в файле local.conf каталога сборки (например, ~/poky/build/conf/local.conf), указывается переменная вида PACKAGE_CLASSES ?= “package_packageformat”, где packageformat может включать значении ipk, rpm, deb и tar в соответствии с поддерживаемыми типами пакетов. Поскольку YP поддерживает четыре формата, в переменной можно указать несколько значений, однако система сборки OE будет применять лишь первый при создании образа или SDK.

Если нужно иметь базовые данные о пакетах в текущей сборке, а также иметь на платформе соответствующий инструмент управления пакетами при работе, можно включить значение package-management в переменную IMAGE_FEATURES. Однако делать это совсем не обязательно и можно создать образ без баз данных и добавить лишь инструменты для управления пакетами при работе. Например, можно включить opkg в переменную IMAGE_INSTALL, если планируется использовать пакеты IPK. Потом можно будет инициализировать базы данных на целевой системе.

Всякий раз, когда при сборке может быть создан новый пакет или изменён существующий, рекомендуется обновить индекс пакетов после сборки с помощью команды bitbake package-index. Можно возникнуть желание обновить индекс пакетов при сборке того или иного пакета с помощью bitbake some-package package-index, но лучше этого не делать, поскольку BitBake не планирует обновление индекса по завершении сборки пакета и не обеспечивается гарантии того, что собранный пакет будет включён в индекс. Лучше обновлять индекс отдельной командой после сборки пакетов.

Можно использовать переменные PACKAGE_FEED_ARCHS, PACKAGE_FEED_BASE_PATHS и PACKAGE_FEED_URIS для настройки в целевом образе работы с хранилищем пакетов. Если этого не сделать, потребуется выполнить инициализацию работы с хранилищем непосредственно на платформе, как описано ниже. Переменные для их корректной работы нужно устанавливать до сборки образа.

По завершении сборки пакеты размещаются в каталоге ${TMPDIR}/deploy/packageformat. Например, при ${TMPDIR} = tmp и выборе формата RPM пакеты RPM будут доступны в каталоге tmp/deploy/rpm.

3.22.4.2. Организация сервера

Обычно для работы с пакетами применяют сервер HTTP, который может работать на основе Apache 2, lighttpd, SimpleHTTPServer и т. п. Для простоты здесь описана организация сервера на основе SimpleHTTPServer, хотя это и не является лучшим решением для производственной среды.

Сервер можно запустить из каталога сборки, где хранятся файлы выбранного варианта пакетов (PACKAGE_CLASSES). Ниже приведён пример с каталогом ~/poky/build/tmp/deploy/rpm, где PACKAGE_CLASSES имеет значение package_rpm.

```
$ cd ~/poky/build/tmp/deploy/rpm
$ python -m SimpleHTTPServer
```

3.22.4.3. Установка цели

Настройка целевой платформы зависит от выбранного менеджера пакетов. Здесь даны сведения для RPM, IPK и DEB.

3.22.4.3.1. Использование RPM

Пакет Dandified Packaging Tool (DNF) обеспечивает управление пакетами RPM в процессе работы. Для этого требуется выполнить начальную установку, если переменные PACKAGE_FEED_ARCHS, PACKAGE_FEED_BASE_PATHS и PACKAGE_FEED_URIS не были установлены до сборки целевого образа.

На целевой платформе нужно уведомить DNF о доступности баз данных о пакетах путём создания файла /etc/yum.repos.d/oe-packages.repo и указания пакетов. Предположим в качестве примера возможность использования на устройстве пакетов i586 и qemux86 с сервера my.server. Важно, чтобы идентификаторы URI в конфигурации репозиториев на платформе корректно указывали удалённые хранилища. Для разработки можно использовать машину сборки, но в рабочей среде лучше вынести пакеты за пределы области сборки и указать их местоположение. Это предотвратит ситуации переписывания системой сборки пакетов в репозитории.

При указании DNF местоположения баз данных о пакетах нужно отдельно указать место для каждой архитектуры или общее место для всех пакетов. Совмещение приведённых ниже вариантов недопустимо.

- *Создание явного списка архитектур* путём определения базовых URL для местоположения пакетов:

  ```
  [oe-packages]
  baseurl=http://my.server/rpm/i586 http://my.server/rpm/qemux86 http://my.server/rpm/all
  ```

  Это указывает DNF базы данных для пакетов трёх вариантов архитектуры.

- *Создание одного (полного) индекса пакетов* с одним URL для базы данных обо всех пакетах.

  ```
  [oe-packages]
  baseurl=http://my.server/rpm
  ```

  Это указывает DNF единую базу данных, содержащую информацию о пакетах для каждой архитектуры.

После указания базы данных можно собрать информацию о пакетах с помощью команды

```
# dnf makecache
```

Получив информацию, DNF может находить, устанавливать и обновлять пакеты из репозиториев. Дополнительную информацию можно найти в документации DNF.

3.22.4.3.2. Использование IPK

Приложение opkg выполняет управление пакетами IPK при работе системы. Для этого требуется выполнить начальную установку, если переменные PACKAGE_FEED_ARCHS, PACKAGE_FEED_BASE_PATHS и PACKAGE_FEED_URIS не были установлены до сборки целевого образа. Программа opkg использует файлы конфигурации для поиска доступных баз данных о пакетах, поэтому нужно создать эти файлы в каталоге /etc/opkg/ для указания репозиториев.

Предположим, например, что обслуживаются пакеты из каталога ipk/, содержащего базы данных для архитектуры i586, all и qemux86, через сервер HTTPmy.server. На целевой платформе файл конфигурации (например, my_repo.conf) в каталоге /etc/opkg/ будет иметь вид

```
src/gz all http://my.server/ipk/all
src/gz i586 http://my.server/ipk/i586
src/gz qemux86 http://my.server/ipk/qemux86
```

После этого выполняется команда извлечения данных из репозитория

```
# opkg update
```

Получив информацию, opkg может находить, устанавливать и обновлять пакеты из репозиториев.

3.22.4.3.3. Использование DEB

Приложение apt выполняет управление пакетами DEB при работе системы. Для этого требуется выполнить начальную установку, если переменные PACKAGE_FEED_ARCHS, PACKAGE_FEED_BASE_PATHS и PACKAGE_FEED_URIS не были установлены до сборки целевого образа.

Для информирования apt о репозитории можно создать файл со списком (например, my_repo.list) в каталоге /etc/apt/sources.list.d/. Например, для обслуживания пакетов из каталога deb/ с базами данных для i586, all, qemux86 с помощью сервера HTTP my.server в список следует включить приведённые ниже строки.

```
deb http://my.server/deb/all ./
deb http://my.server/deb/i586 ./
deb http://my.server/deb/qemux86 ./
```

После этого выполняется команда извлечения данных из репозитория

```
# apt-get update
```

Получив информацию, apt может находить, устанавливать и обновлять пакеты из репозиториев.

3.22.5. Создание и использование подписанных пакетов

Для улучшения защиты пакетов RPM можно использовать подписи. После проверки подписи система сборки OE может использовать пакет, но если подпись не соответствует, сборка будет прервана.

3.22.5.1. Подписывание пакетов RPM

Для подписывания пакетов RPM нужно добавить в файл local.conf или distro.conf приведённые ниже строки.

```
# Inherit sign_rpm.bbclass to enable signing functionality
INHERIT += " sign_rpm"
# Define the GPG key that will be used for signing.
RPM_GPG_NAME = "key_name"
# Provide passphrase for the key
RPM_GPG_PASSPHRASE = "passphrase"
```

Вместо key_name и passphrase следует указать действительное имя ключа и пароль. Кроме указанных переменных RPM_GPG_NAME и RPM_GPG_PASSPHRASE с подписями связаны две необязательных переменных:

- *GPG_BIN* задаёт двоичный файл gpg, выполняемый при подписывании пакета;
- *GPG_PATH* указывает домашний каталог gpg, используемый при подписывании пакета.

3.22.5.2. Подписывание хранилища пакетов

Можно организовать подписанные хранилища для пакетов IPK и RPM. Для этого нужно включить в local.conf или distro.conf приведённые ниже строки.

```
INHERIT += "sign_package_feed"
PACKAGE_FEED_GPG_NAME = "key_name"
PACKAGE_FEED_GPG_PASSPHRASE_FILE = "path_to_file_containing_passphrase"
```

Для хранилища подписанных пакетов парольная фраза должна храниться в отдельном файле, указанном переменной PACKAGE_FEED_GPG_PASSPHRASE_FILE. Вынос пароля за пределы конфигурации улучшает защиту.

Кроме переменных PACKAGE_FEED_GPG_NAME и PACKAGE_FEED_GPG_PASSPHRASE_FILE есть три необязательных переменных, относящихся к подписыванию пакетов:

- *GPG_BIN* задаёт двоичный файл gpg, выполняемый при подписывании пакета;
- *GPG_PATH* указывает домашний каталог gpg, используемый при подписывании пакета;
- *PACKAGE_FEED_GPG_SIGNATURE_TYPE* задаёт тип подписи gpg. Переменная применима лишь к хранилищам пакетов RPM и IPK, а возможные значения включают ASC (принято по умолчанию) и BIN.

3.22.6. Тестирование пакетов с помощью ptest

Проверка пакетов (ptest) работает с пакетами, созданными системой сборки OE, на целевой платформе и содержит по меньшей мере два элемента - реальный тест и shell-сценарий (run-ptest) для запуска теста. Включение теста в сценарий запуска не разрешается. Сам тест может быть любым - от shell-сценария, запускающего двоичный файл и проверяющего результат, до сложной системы двоичных файлов для тестирования и файлов данных.

Вывод таста имеет формат, применяемый Automake, - result: testname, где result может принимать значение PASS, FAIL или SKIP, а testname может быть строкой идентификации. Список заданий YP, для которых уже включён ptest, приведён на странице Ptest. Такие задания наследуют класс ptest.

3.22.6.1. Добавление ptest в сборку
Для добавления тестирования пакетов в сборку нужно задать DISTRO_FEATURES и EXTRA_IMAGE_FEATURES в файле local.conf в каталоге сборки, как показано ниже.

```
DISTRO_FEATURES_append = " ptest"
EXTRA_IMAGE_FEATURES += "ptest-pkgs"
```

По завершении сборки файлы ptest размещаются в каталоге /usr/lib/package/ptest внутри образа (package - имя пакета).

3.22.6.2. Запуск ptest
Пакет ptest-runner устанавливает сценарий, который выполняет все установленные наборы тестов ptest в заданном порядке. Имеет смысл включать этот пакет в образ.

3.22.6.3. Подготовка пакета
Чтобы включить тесты ptest на целевом оборудовании, нужно подготовить задания к сборке.

- *Наследование класса ptest* требует включения в задание строки inherit ptest.
- *Создание сценария run-ptest* для запуска теста. Сценарий указывается в переменной SRC_URI. Например, для запуска теста dbus сценарий может иметь вид

```
#!/bin/sh
cd test
make -k runtest-TESTS
```

- *Контроль соблюдения зависимостей.* Если тест добавляет зависимости при сборке или работе, которых обычно нет для пакета (например, запуск make при тестировании), нужно использовать переменные DEPENDS и RDEPENDS в задании для указания зависимостей. Например зависимость от make при работе задаётся строкой RDEPENDS_${PN}-ptest += "make".
- *Добавление функции для сборки теста.* Не все пакеты поддерживают кросс-компиляцию своих тестов и может потребоваться добавление функции кросс-компиляции в пакет. Многие пакеты на основе Automake компилируют и запускают свои тесты с использованием одной команды, такой как make check. Однако на хосте сборки эта команда создаст и запустит тесты локально, тогда как кросс-компиляция требует сборки пакета на хосте и запуска на целевой системе. Версия Automake из состава YP включает исправления, разделяющие сборку и выполнение. Поэтому для пакетов кросс-компиляция make check выполняется автоматически. Тем не менее, все равно нужно добавить функцию do_compile_ptest для сборки теста в задание.

```
do_compile_ptest() {
   oe_runmake buildtest-TESTS
}
```

- *Установка нужной конфигурации* (если она нужна для компиляции тестового кода) с помощью функции do_configure_ptest в задании.
- *Установка тестов.* Класс ptest автоматически копирует run-ptest в целевую систему, а затем запускает make install-ptest для инициирования тестов. Если этого не достаточно, нужно создать функцию do_install_ptest и обеспечить её вызов по завершении make install-ptest.

3.22.7. Создание пакетов NPM
NPM является менеджером пакетов для JavaScript. YP поддерживает сборщик NPM (fetcher), который можно применять с devtool для подготовки заданий, создающих пакеты NPM. Имеется два метода создания пакетов NPM с помощью devtool - реестр модулей NPM и код проекта NPM. Можно создавать задания NPM вручную, но с devtool это проще.

3.22.7.1. Требования и предостережения
Перед использованием devtool для создания пакетов NPM нужно выполнить указанные ниже действия.

- Можно использовать один из двух методов создания пакетов NPM и подход с реестром несколько проще. Однако можно рассмотреть и вариант с проектом поскольку модель может не публиковаться в общедоступном реестре NPM (npm-registry).
- Следует ознакомиться с devtool.
- На хосте сборки нужен пакет nodejs-npm, являющийся частью среды OE, который можно получить клонированием репозитория https://github.com/openembedded/meta-openembedded. Путь к локальной копии нужно указать в файле bblayers.conf.
- devtool не может обнаруживать естественные библиотеки в зависимостях модуля, поэтому нужно вручную добавлять пакеты в задание.
- При развёртывании пакетов NPM devtool не может определить отсутствие нужного пакета на целевой платформе (например, nodejs), поэтому нужно проверять самостоятельно.
- Хотя NPM может не требоваться для пакета, лучше иметь NPM (nodejs-npm) на целевой платформе.

3.22.7.2. Реестр модулей NPM
Здесь рассматривается пример модуля cute-files, который является программой web-браузера. Нужно знать версию модуля. Первым делом нужно использовать devtool и сборщик NPM для подготовки задания

```
$ devtool add "npm://registry.npmjs.org;name=cute-files;version=1.0.2"
```

Команда devtool add запускает recipetool create и использует тот же URI выборки для загрузки каждой зависимости и деталей лицензирования, когда это возможно. Файл задания достаточно прост и содержит все лицензии, найденные recipetool, а также лицензии из переменных LIC_FILES_CHKSUM. Нужно проверить переменные на предмет значений unknown в поле LICENSE и добавить соответствующую информацию вручную.

Команда recipetool создаёт файлы shrinkwrap и lockdown для задания. Файлы shrinkwrap включают версии всех зависимых модулей, но многие пакеты не включают файлов shrinkwrap и recipetool создаёт их. Можно заменить файл shrinkwrap своим файлом, задав переменную NPM_SHRINKWRAP. Файлы lockdown содержат контрольные суммы каждого модуля, чтобы определить загрузку тех же файлов при сборке задания. Файлы lockdown обеспечивают сохранение зависимостей и сохранение в реестре NPM того же файла.

Пакет создаётся для каждого субмодуля. Это правило обеспечивает единственный практичный способ получения лицензий для всех зависимостей, представленных в манифесте лицензий образа. Команда devtool edit-recipe позволяет просматривать задание.

```
$ devtool edit-recipe cute-files
SUMMARY = "Turn any folder on your computer into a cute file browser, available on the local network."
LICENSE = "BSD-3-Clause & Unknown & MIT & ISC"
LIC_FILES_CHKSUM = "file://LICENSE;md5=71d98c0a1db42956787b1909c74a86ca \
 file://node_modules/content-disposition/LICENSE;md5=c6e0ce1e688c5ff16db06b7259e9cd20 \
 file://node_modules/express/LICENSE;md5=5513c00a5c36cd361da863dd9aa8875d \
 ...

SRC_URI = "npm://registry.npmjs.org;name=cute-files;version=${PV}"
NPM_SHRINKWRAP := "${THISDIR}/${PN}/npm-shrinkwrap.json"
NPM_LOCKDOWN := "${THISDIR}/${PN}/lockdown.json"
inherit npm
# Must be set after inherit npm since that itself sets S
S = "${WORKDIR}/npmpkg"

LICENSE_${PN}-content-disposition = "MIT"
...
LICENSE_${PN}-express = "MIT"
LICENSE_${PN} = "MIT"
```

В примере следует отметить 3 важных аспекта:

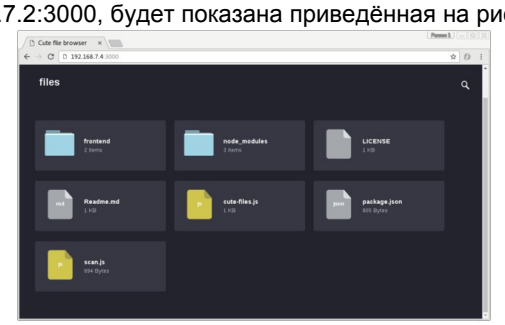
- SRC_URI применяет схему NPM, поэтому используется сборщик NPM;
- recipetool собирает данные о всех лицензиях и при отсутствии лицензии для субмодуля перед его именем помещается символ комментария;
- оператор inherit npm заставляет класс npm упаковывать все модули.

Для сборки пакета cute-files можно воспользоваться командой devtool build cute-files. Следует помнить, что nodejs нужно установить на целевой платформе до пакета. Предположим, что целевая система имеет адрес 192.168.7.2 и введём команду devtool deploy-target -s cute-files root@192.168.7.2 для установки пакета, после чего можно будет протестировать приложение.

Известные проблемы не позволяют просто запустить cute-files как при запуске npm install.

```
$ cd /usr/lib/node_modules/cute-files
$ node cute-files.js
```

Если в браузере ввести http://192.168.7.2:3000, будет показана приведённая на рисунке страница.



Задание можно найти в каталоге workspace/recipes/cute-files и использовать на любом уровне.

3.22.7.3. Код проекта NPM

Хотя полезно упаковывать модули уже в реестре NPM, добавление проектов node.js более распространено. Этот метод очень похож на использование реестра и команде devtool предоставляется URL исходных файлов. Используя прошлый пример с cute-files, вводим команду devtool add https://github.com/martinaglv/cute-files.git. Эта команда создаёт задание, очень похожее на задание из предыдущего параграфа, однако переменная SRC_URI имеет вид

```
   SRC_URI = "git://github.com/martinaglv/cute-files.git;protocol=https \
  npm://registry.npmjs.org;name=commander;version=2.9.0;subdir=node_modules/commander \
  npm://registry.npmjs.org;name=express;version=4.14.0;subdir=node_modules/express \
  npm://registry.npmjs.org;name=content-disposition;version=0.3.0;subdir=node_modules/content-disposition \
  "
```

Здесь основной модуль берётся из репозитория Git, а зависимости - из реестра NPM. В остальном задания совпадают Можно собрать и развернуть пакет в соответствии с предыдущим параграфом.

3.23. Эффективная выборка файлов при сборке

Система сборки OE работает с файлами исходного кода, указанными переменной SRC_URI. При сборке с использованием BitBake значительную часть работы составляет нахождение и загрузка всех архивов источников. Для

образов это может занимать много времени. В этом разделе описано использование зеркал для ускорения выборки исходных файлов, а также предварительная выборка файлов.

3.23.1. Установка эффективных зеркал
В сборке YP можно просто загрузить все архивы исходного кода. При наличии исходных кодов из других систем сборки можно обеспечить существенную экономию времени, добавив каталоги с этим кодом в переменные системы сборки. Ниже приведён вариант указания таких файлов в конфигурационном файле local.conf.

```
SOURCE_MIRROR_URL ?= "file:///home/you/your-download-dir/"
INHERIT += "own-mirrors"
BB_GENERATE_MIRROR_TARBALLS = "1"
# BB_NO_NETWORK = "1"
```

Здесь переменная BB_GENERATE_MIRROR_TARBALLS заставляет систему сборки OE создавать файлы репозиториев Git и сохранять их в каталоге DL_DIR. Из соображений производительности такое поведение системы сборки не задано по умолчанию. Можно также использовать переменную PREMIRRORS [3].

3.23.2. Подготовка исходных файлов без сборки
Другим вариантом является предварительная выборка исходных кодов без запуска сборки. Это позволяет избежать проблем с загрузкой и собрать весь исходный код в каталоге build/downloads внутри DL_DIR. Сделать это можно с помощью команды bitbake -c target runall="fetch". Этот вариант гарантирует наличие всех исходных кодов и возможность сборки даже без доступа в Internet.

3.24. Выбор менеджера инициализации
По умолчанию YP использует менеджер инициализации SysVinit, однако поддерживается и менеджер systemd, который является полной заменой init с параллельным запуском служб, сниженной загрузкой оболочки и другими функциями. Для использования SysVinit ничего делать не нужно, но для systemd потребуются некоторые шаги, описанные ниже.

3.24.1. Использование только systemd
В файле конфигурации дистрибутива следует указать

```
DISTRO_FEATURES_append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
```

В первой строке могут присутствовать и другие свойства. Можно также отключить SysVinit с помощью DISTRO_FEATURES_BACKFILL_CONSIDERED += "sysvinit". В результате этого будут удалены ненужные сценарии SysVinit. Для полного удаления сценариев инициализации следует указать VIRTUAL-RUNTIME_initscripts = "".

3.24.2. Systemd для основного образа и SysVinit для восстановления
В этом случае следует включить в файл конфигурации дистрибутива приведённые ниже переменные.

```
DISTRO_FEATURES_append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
```

В этом случае основной образ будет использовать задание packagegroup-core-boot.bb и systemd, а восстановительный (минимальный) не будет применять указанную группу пакетов. Однако можно установить SysVinit и пакеты, поддерживающие systemd и SysVinit.

3.25. Выбор менеджера устройств
YP обеспечивает несколько способов управления менеджером устройств (/dev).

- *Постоянный и заранее заполненный каталог /dev*. В этом случае каталог /dev существует всегда и требуется создание узлов для устройств в процесс сборки.
- *Использование devtmpfs с менеджером устройств.* В этом случае каталог /dev обеспечивается ядром как файловая система в памяти и автоматически заполняется ядром при работе. Дополнительная настройка узлов для устройств выполняется в пользовательском пространстве менеджером устройств, таким как udev или busybox-mdev.

3.25.1. Использование постоянного заполненного каталога /dev
Для использования статического метода заполнения устройств нужно задать USE_DEVFS = "0". Содержимое получаемого каталога /dev определяется файлом с таблицей устройств. Переменная IMAGE_DEVICE_TABLES задаёт используемую таблицу и должна быть установлена в файле конфигурации машины или дистрибутива, но можно задать её и в файле local.conf. По умолчанию применяется IMAGE_DEVICE_TABLES = "device_table-mymachine.txt". Заполнение каталога обеспечивается утилитой makedevs в процессе сборки образа.

3.25.2. Использование devtmpfs и менеджера устройств
Для динамического заполнения устройств нужно использовать USE_DEVFS = "1". При установленной переменной каталог /dev заполняется ядром с помощью devtmpfs, для чего в конфигурации ядра должна быть установлена переменная CONFIG_DEVTMPFS. Созданные devtmpfs устройства принадлежат пользователю root и имеют права доступа 0600.

Для управления узлами устройств можно использовать такие менеджеры устройств, как udev или busybox-mdev. Выбор менеджера задаётся переменной VIRTUAL-RUNTIME_dev_manager в файле конфигурации машины или дистрибутива или в файле local.conf, как показано ниже.

```
VIRTUAL-RUNTIME_dev_manager = "udev"

# Some alternative values
# VIRTUAL-RUNTIME_dev_manager = "busybox-mdev"
```

```
# VIRTUAL-RUNTIME_dev_manager = "systemd"
```

3.26. Использование внешних SCM

При работе с заданием из внешней системы SCM система сборки OE может уведомлять об изменении заданий в SCM и соберёт в результате пакеты, зависящие от новых заданий, используя последнюю версию. Это работает лишь с SCM, где можно получить номер выпуска для изменений. В настоящее время это возможно для репозиториев Apache Subversion (SVN), Git и Bazaar (BZR). Для включения такого режима следует указать в переменной задания PV значение SRCPV, например, PV = "1.2.3+git${SRCPV}", а в файл local.conf нужно добавить строку SRCREV_pn-PN = "${AUTOREV}". PN - это имя задания, для которого нужно включить автоматическое обновление исходного кода. Можно не менять файл local.conf, а добавить в задание строку SRCREV = "${AUTOREV}".

YP включает дистрибутив poky-bleeding с файлом конфигурации, содержащим строку

```
require conf/distro/include/poky-floating-revisions.inc
```

Эта строка добавляет включаемый файл с множеством строк вида

```
#SRCREV_pn-opkg-native ?= "${AUTOREV}"
#SRCREV_pn-opkg-sdk ?= "${AUTOREV}"
#SRCREV_pn-opkg ?= "${AUTOREV}"
#SRCREV_pn-opkg-utils-native ?= "${AUTOREV}"
#SRCREV_pn-opkg-utils ?= "${AUTOREV}"
SRCREV_pn-gconf-dbus ?= "${AUTOREV}"
SRCREV_pn-matchbox-common ?= "${AUTOREV}"
SRCREV_pn-matchbox-config-gtk ?= "${AUTOREV}"
SRCREV_pn-matchbox-desktop ?= "${AUTOREV}"
SRCREV_pn-matchbox-keyboard ?= "${AUTOREV}"
SRCREV_pn-matchbox-panel-2 ?= "${AUTOREV}"
SRCREV_pn-matchbox-themes-extra ?= "${AUTOREV}"
SRCREV_pn-matchbox-terminal ?= "${AUTOREV}"
SRCREV_pn-matchbox-wm ?= "${AUTOREV}"
SRCREV_pn-settings-daemon ?= "${AUTOREV}"
SRCREV_pn-screenshot ?= "${AUTOREV}"
     ...
```

Это позволяет создать дистрибутив с отслеживанием обновления исходного кода для многих пакетов. Однако дистрибутив poky-bleeding не тестируется как другие дистрибутивы и применять его следует с осторожностью.

3.27. Создание корневой файловой системы лишь для чтения

Из соображений безопасности может потребоваться запрет операций записи в корневой файловой системе целевого устройства. Возможны также устройства хранения, не поддерживающие запись. В таких случаях нужно обеспечить соответствующее поведение образа. Для поддержки файловых систем без возможности записи требуется предотвращение попыток записи операционной системы и приложений в корневую файловую систему. Для этого нужно настроить все компоненты целевой системы на запись в другое место или аккуратно предотвращать попытки записи в корневую файловую систему.

3.27.1. Создание корневой файловой системы

Для создания файловой системы без возможности записи к образу просто добавляется свойство read-only-rootfs. Для этого можно указать в задании для образа или в файле local.conf каталога сборки строку IMAGE_FEATURES = "read-only-rootfs" или EXTRA_IMAGE_FEATURES += "read-only-rootfs". Дополнительная информация об использовании этих переменных дана в параграфе 3.2.2. Настройка с помощью IMAGE_FEATURES и EXTRA_IMAGE_FEATURES и описаниях переменных IMAGE_FEATURES и EXTRA_IMAGE_FEATURES.

3.27.2. Сценарии пост-установки

Важно обеспечить выполнение сценариев пост-установки (pkg_postinst) для пакетов, включённых в образ, при создании корневой файловой системы при сборке на хосте, поскольку эти сценарии не смогут работать при первой загрузке на целевой системе. При включённом свойстве read-only-rootfs система сборки проверяет в процессе создания файловой системы выполнение всех сценариев пост-утсановки. Если какой-то из этих сценариев требует запуска после создания файловой системы, сборка завершается отказом. Проверка во время сборки гарантирует отказ сборки, а не первой загрузки на целевой системе.

Большинство сценариев пост-установки, создаваемых системой сборки из состава YP, созданы с возможностью выполнения в процессе создания файловой системы (например, сценарий пост-установки для кэширования шрифтов). Однако при создании своих сценариев нужно обеспечить возможность их работы при создании файловой системы.

Ниже указаны некоторые общие проблемы, возникающие при работе сценариев пост-установки.

- *Не используется $D в начале абсолютного пути.* Система сборки определяет $D при создании корневой файловой системы, а при запуске на целевом устройстве значение $D пусто. Это предполагает двойное назначение $D - обеспечения корректности путей на хосте сборки и в целевой системе, а также определение используемой среды для выполнения соответствующих действий.
- *Попытка запуска процессов, относящихся к целевой архитектуре или зависимых от неё.* Это можно обойти путём использования естественных инструментов, работающих на хосте сборки для решения тех же задач, или запуска процессов в QEMU с функцией qemu_run_binary.

3.27.3. Области с доступом для записи

При включённом свойстве read-only-rootfs любая попытка записи в корневую файловую систему будет давать отказ. Поэтому нужно обеспечить запись процессов и приложений в иную область, например, /tmp или /var/run.

3.28. Поддержка качества вывода сборки

На качество сборки может влиять много факторов. Например, обновление версии задания из восходящего источника или эксперимент с опциями конфигурации могут привести к незаметным изменениям, которые проявятся позже. В случае задания, использующего обновлённую версию программы это может вносить необязательную зависимость от другой библиотеки, нежели была определена автоматически. Если такая библиотека уже собрана, программа будет связана с ней и библиотека будет добавлена в образ вместе с новой программой, даже если вы не хотите её использовать.

Класс buildhistory помогает поддерживать качество вывода сборки и его можно использовать для выявления неожиданных и возможно нежелательных изменений в выводе. При включении истории сборки записываются данные о содержимом каждого пакета и образа, а затем информация фиксируется в локальном репозитории Git, где можно её проверить.

3.28.1. Управление историей сборки

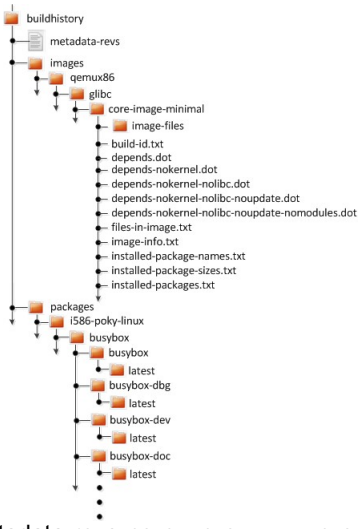
История сборки выключена по умолчанию и для включения нужно добавить оператор INHERIT и установить переменную BUILDHISTORY_COMMIT в конце файла conf/local.conf в каталоге сборки.

```
INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "1"
```

Включение истории сборки заставляет систему сборки OE собирать выходную информацию и представлять её в форме одной фиксации (commit) в локальный репозиторий Git. Включение истории сборки несколько увеличивает время сборки и занимаемое на диске пространство.

3.28.2. Содержимое истории сборки

История сборки хранится в каталоге ${TOPDIR}/buildhistory внутри каталога сборки, как определено в переменной BUILDHISTORY_DIR. На рисунке представлена сокращённая структура истории сборки.



На верхнем уровне размещается файл metadata-revs со списком выпусков репозиториев для включённых при сборке уровней. Остальные данные поделены на отдельные каталоги для пакетов, образов и sdk, как описано ниже.

3.28.2.1. История сборки пакета

История сборки для каждого пакета включает текстовый файл с парами «имя-значение». Например, файл buildhistory/packages/i586-poky-linux/busybox/busybox/latest содержит приведённые ниже данные.

```
PV = 1.22.1
PR = r32
RPROVIDES =
RDEPENDS = glibc (>= 2.20) update-alternatives-opkg
RRECOMMENDS = busybox-syslog busybox-udhcpc update-rc.d
PKGSIZE = 540168
FILES = /usr/bin/* /usr/sbin/* /usr/lib/busybox/* /usr/lib/lib*.so.* \
   /etc /com /var /bin/* /sbin/* /lib/*.so.* /lib/udev/rules.d \
   /usr/lib/udev/rules.d /usr/share/busybox /usr/lib/busybox/* \
   /usr/share/pixmaps /usr/share/applications /usr/share/idl \
   /usr/share/omf /usr/share/sounds /usr/lib/bonobo/servers
FILELIST = /bin/busybox /bin/busybox.nosuid /bin/busybox.suid /bin/sh \
   /etc/busybox.links.nosuid /etc/busybox.links.suid
```

Большинство пар соответствует переменным, использованным для создания пакета. Исключением является переменная FILELIST со списком реальных файлов из пакета и PKGSIZE с общим размером файлов пакета в байтах.

Имеется также файл, соответствующий заданию, из которого получен пакет (например, buildhistory/packages/i586-poky-linux/busybox/latest).

```
PV = 1.22.1
PR = r32
DEPENDS = initscripts kern-tools-native update-rc.d-native \
   virtual/i586-poky-linux-compilerlibs virtual/i586-poky-linux-gcc \
   virtual/libc virtual/update-alternatives
PACKAGES = busybox-ptest busybox-httpd busybox-udhcpd busybox-udhcpc \
   busybox-syslog busybox-mdev busybox-hwclock busybox-dbg \
   busybox-staticdev busybox-dev busybox-doc busybox-locale busybox
```

Для заданий, извлечённых из систем контроля версий (например, Git), имеется файл со списком выпусков исходного кода, указанных в задании, и списком реальных выпусков, использованных для сборки. Эти списки могут различаться при установке SRCREV = ${AUTOREV}. Ниже приведён пример из файла buildhistory/packages/qemux86-poky-linux/linux-yocto/latest_srcrev.

```
# SRCREV_machine = "38cd560d5022ed2dbd1ab0dca9642e47c98a0aa1"
SRCREV_machine = "38cd560d5022ed2dbd1ab0dca9642e47c98a0aa1"
# SRCREV_meta = "a227f20eff056e511d504b2e490f3774ab260d6f"
SRCREV_meta = "a227f20eff056e511d504b2e490f3774ab260d6f"
```

Можно использовать команду buildhistory-collect-srcrevs с опцией -a для получения сохранённых значений SRCREV из истории сборки в формате, пригодном для использования в глобальной конфигурации (например, local.conf или включаемый файл дистрибутива) с целью переопределения значений AUTOREV фиксированным набором выпусков. Пример вывода представлен ниже.

```
$ buildhistory-collect-srcrevs -a
# i586-poky-linux
SRCREV_pn-glibc = "b8079dd0d360648e4e8de48656c5c38972621072"
SRCREV_pn-glibc-initial = "b8079dd0d360648e4e8de48656c5c38972621072"
SRCREV_pn-opkg-utils = "53274f087565fd45d8452c5367997ba6a682a37a"
SRCREV_pn-kmod = "fd56638aed3fe147015bfa10ed4a5f7491303cb4"
# x86_64-linux
SRCREV_pn-gtk-doc-stub-native = "1dea266593edb766d6d898c79451ef193eb17cfa"
SRCREV_pn-dtc-native = "65cc4d2748a2c2e6f27f1cf39e07a5dbabd80ebf"
SRCREV_pn-update-rc.d-native = "eca680ddf28d024954895f59a241a622dd575c11"
SRCREV_glibc_pn-cross-localedef-native = "b8079dd0d360648e4e8de48656c5c38972621072"
SRCREV_localedef_pn-cross-localedef-native = "c833367348d39dad7ba018990bfdaffaec8e9ed3"
SRCREV_pn-prelink-native = "faa069deec99bf61418d0bab831c83d7c1b797ca"
SRCREV_pn-opkg-utils-native = "53274f087565fd45d8452c5367997ba6a682a37a"
SRCREV_pn-kern-tools-native = "23345b8846fe4bd167efdf1bd8a1224b2ba9a5ff"
SRCREV_pn-kmod-native = "fd56638aed3fe147015bfa10ed4a5f7491303cb4"
# qemux86-poky-linux
SRCREV_machine_pn-linux-yocto = "38cd560d5022ed2dbd1ab0dca9642e47c98a0aa1"
SRCREV_meta_pn-linux-yocto = "a227f20eff056e511d504b2e490f3774ab260d6f"
# all-poky-linux
SRCREV_pn-update-rc.d = "eca680ddf28d024954895f59a241a622dd575c11"
```

Ниже приведены некоторые замечания по использованию команды buildhistory-collect-srcrevs.

- По умолчанию выводятся только значения, где SRCREV не задана жёстко (обычно при использовании AUTOREV). Опция -a задаёт вывод всех значений SRCREV.
- Выходные операторы могут не давать эффекта, если где-то в конфигурации системы сборки применены переопределения. Опция -f добавляет форсированные переопределения в каждую строку, если нужно обойти это ограничение.
- Сценарий не использует особой обработки при сборке для нескольких машин, однако помещает символ комментария перед каждым набором значений, задающим триплет, который уже был показан (например, i586-poky-linux).

3.28.2.2. История сборки образа

Для каждого образа создаётся набор описанных ниже файлов.

- *image-files* - каталог с выбранными файлами из корневой системы (заданы в BUILDHISTORY_IMAGE_FILES).
- *build-id.txt* - понятные человеку сведения о конфигурации сборки и метаданные выпусков исходного кода, а также полный заголовок сборки, выводимый BitBake.
- **.dot* - графы зависимостей для образа, совместимые с graphviz.
- *files-in-image.txt* - список файлов в образе с правами доступа, владельцами, группами, размером и символьными ссылками.
- *image-info.txt* - текстовый файл с информацией об образе в виде пар «имя-значение».
- *installed-package-names.txt* - список имён установленных пакетов.
- *installed-package-sizes.txt* - список имён установленных пакетов, упорядоченный по размерам.
- *installed-packages.txt* - список имён установленных пакетов с полными именами.

Информация об установленных пакетах может собираться даже при запрете включения менеджера пакетов в финальный образ. Ниже приведён пример файла image-info.txt.

```
DISTRO = poky
DISTRO_VERSION = 1.7
USER_CLASSES = buildstats image-mklibs image-prelink
IMAGE_CLASSES = image_types
IMAGE_FEATURES = debug-tweaks
IMAGE_LINGUAS =
IMAGE_INSTALL = packagegroup-core-boot run-postinsts
BAD_RECOMMENDATIONS =
NO_RECOMMENDATIONS =
PACKAGE_EXCLUDE =
ROOTFS_POSTPROCESS_COMMAND = write_package_manifest; license_create_manifest; \
   write_image_manifest ; buildhistory_list_installed_image ; \
   buildhistory_get_image_installed ; ssh_allow_empty_password;  \
   postinst_enable_logging; rootfs_update_timestamp ; ssh_disable_dns_lookup ;
IMAGE_POSTPROCESS_COMMAND =   buildhistory_get_imageinfo ;
```

```
IMAGESIZE = 6900
```

Размеры файлов указываются в килобайтах (кроме общего размера IMAGESIZE), пары «имя-значение» являются переменными, которые могут влиять на содержимое образа. Эта информация полезна для определения причин изменений в пакетах или файлах.

3.28.2.3. Использование истории сборки для получения информации только об образе

История сборки создаёт информацию об образе включая графы зависимостей, что позволяет увидеть причины добавления компонент в образ. Если нужны лишь сводные данные для образа и не интересуют отдельные пакеты или данные SDK, можно включить запись лишь данных образа без истории, указав в файле conf/local.conf каталога сборки

```
INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "0"
BUILDHISTORY_FEATURES = "image"
```

3.28.2.4. История сборки SDK

Похожая информация собирается в истории и для сборки SDK (например, bitbake -c populate_sdk imagename). Эта информация различается для стандартных и расширяемых SDK. Список файлов истории сборки приведён ниже.

- *files-in-sdk.txt* - список файлов в SDK с правами доступа, владельцами, группами, размером и символьными ссылками для хостовой и целевой части SDK.
- *sdk-info.txt* - текстовый файл с информацией об SDK в виде пар «имя-значение».
- *sstate-task-sizes.txt* - текстовый файл в виде пар «имя-значение» с данными о размерах групп задач (например, do_populate_sysroot), создаваемый только при сборке расширяемых SDK.
- *sstate-package-sizes.txt* - текстовый файл в виде пар «имя-значение» с данными о пакетах с общим состоянием и размерах в SDK, создаваемый только при сборке расширяемых SDK.
- *sdk-files* — каталог с копиями файлов, указанных в BUILDHISTORY_SDK_FILES, если эти файлы имеются в выводе. По умолчанию BUILDHISTORY_SDK_FILES относится к расширяемым SDK, но его можно установить и для стандартных. Файлы conf/local.conf, conf/bblayers.conf, conf/auto.conf, conf/locked-sigs.inc и conf/devtool.conf копируются по умолчанию в этот каталог для расширяемых SDK.
- В каталоги для хоста и цели включаются перечисленные ниже файлы частей SDK, работающих на хосте и цели. Большая часть этих файлов для расширяемого SDK будет пустой, поскольку расширяемые SDK не состоят из пакетов, как стандартные SDK.
  - *depends.dot* - графы зависимостей для SDK, совместимые с graphviz.
  - *installed-package-names.txt* - список имён установленных пакетов.
  - *installed-package-sizes.txt* - список имён установленных пакетов, упорядоченный по размерам.
  - *installed-packages.txt* - список имён установленных пакетов с полными именами.

Ниже представлен пример файла sdk-info.txt.

```
DISTRO = poky
DISTRO_VERSION = 1.3+snapshot-20130327
SDK_NAME = poky-glibc-i686-arm
SDK_VERSION = 1.3+snapshot
SDKMACHINE =
SDKIMAGE_FEATURES = dev-pkgs dbg-pkgs
BAD_RECOMMENDATIONS =
SDKSIZE = 352712
```

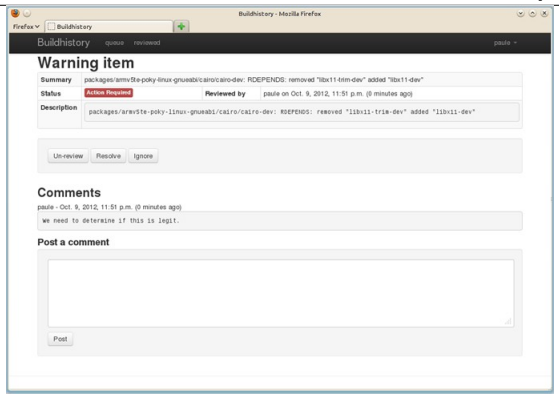
Размеры файлов указываются в килобайтах (кроме общего размера IMAGESIZE), пары «имя-значение» являются переменными, которые могут влиять на содержимое образа. Эта информация полезна для определения причин изменений в пакетах или файлах.

3.28.2.5. Просмотр данных истории сборки

Историю сборки можно посмотреть с помощью команды или через web-интерфейс. Для просмотра изменений (в предположении BUILDHISTORY_COMMIT = "1") можно использовать обычную команду Git, например, git log -p. Следует помнить, что этот метод показывает и несущественные изменений (например, размер пакета). Имеется команда buildhistory-diff, которая обращается к репозиторию Git и выводит существенные различия в удобной форме. Например,

```
$ ~/poky/poky/scripts/buildhistory-diff . HEAD^
Changes to images/qemux86_64/glibc/core-image-minimal (files-in-image.txt):
   /etc/anotherpkg.conf was added
   /sbin/anotherpkg was added
   * (installed-package-names.txt):
   *   anotherpkg was added
Changes to images/qemux86_64/glibc/core-image-minimal (installed-package-names.txt):
   anotherpkg was added
packages/qemux86_64-poky-linux/v86d: PACKAGES: added "v86d-extras"
   * PR changed from "r0" to "r1"
   * PV changed from "0.1.10" to "0.1.12"
packages/qemux86_64-poky-linux/v86d/v86d: PKGSIZE changed from 110579 to 144381 (+30%)
   * PR changed from "r0" to "r1"
   * PV changed from "0.1.10" to "0.1.12"
```

Для buildhistory-diff нужен пакет GitPython, который можно установить с помощью команды pip3 install GitPython --user или менеджера пакетов (пакет python3-git). Просмотр изменений истории через web-интерфейс описан в файле README (http://git.yoctoproject.org/cgit/cgit.cgi/buildhistory-web/). Пример вывода показан на рисунке.



3.29. Автоматизированное тестирование при работе

Система сборки OE обеспечивает автоматизированные тесты для образов, позволяющие проверить функциональность при работе. Тесты можно запустить в QEMU или на реальном оборудовании. Тесты написаны на языке Python с применениям модуля unittest и большинство из них запускают команды на целевой системе через SSH. Сведения о тестах и инфраструктуре QA в составе YP приведены в разделе Testing and Quality Assurance [3].

3.29.1. Включение тестов

Управление тестами различается для случаев QEMU и реального оборудования, как описано ниже.

3.29.1.1. Включение тестов при выполнении в QEMU

Для запуска тестов нужно выполнить перечисленные ниже операции.

- *Настройка для работы через сеть без sudo.*
  - Добавить NOPASSWD для нужного пользователя в файл /etc/sudoers для всех команд или runqemu-ifup. Нужно указать полный путь, поскольку он может меняться для разных копий репозитория источников.
  - В некоторых дистрибутивах нужно также «закомментировать» строку Defaults requiretty в /etc/sudoers.
  - Вручную настроить интерфейс tap в системе.
  - Запустить от имени root сценарий scripts/runqemu-gen-tapdevs, который должен создать список устройств tap. Обычно эта опция выбирается для сред Autobuilder.
    - Обеспечить использование абсолютного пути при вызове сценария с sudo.
    - Для работы сценария нужно собрать задание qemu-helper-native командой bitbake qemu-helper-native.
- *Установка переменной DISPLAY* для работы с X-сервером (например, vncserver для машин без монитора).
- *Настройка на межсетевом экране входящих соединений из сети 192.168.7.0/24.* Некоторые тесты (например, DNF) запускают сервер HTTP на случайном порту с большим номером, используемый для работы с файлами на целевой системе. Модуль DNF обслуживает ${WORKDIR}/oe-rootfs-repo, поэтому он может запускать команды DNF, требующие, чтобы межсетевой экран хоста пропускал входящие соединения из сети 192.168.7.0/24, которая по умолчанию используется runqemu для устройств tap.
- *Проверка наличия на хосте нужных пакетов*:
  - Ubuntu и Debian - sysstat и iproute2;
  - OpenSUSE - sysstat и iproute2;
  - Fedora - sysstat и iproute;
  - CentOS - sysstat и iproute.

После запуска тестов выполняются перечисленные ниже действия.

1. Копия корневой файловой системы записывается в ${WORKDIR}/testimage.
2. Образ загружается в QEMU с помощью стандартного сценария runqemu.
3. По умолчанию используется время ожидания 500 секунд для завершения процесса загрузки и вывода приглашения на вход в систему. Время задаёт переменная TEST_QEMUBOOT_TIMEOUT в файле local.conf.
4. После появления приглашения на вход в систему запускается тест. Полный журнал загрузки будет доступен в каталоге ${WORKDIR}/testimage/qemu_boot_log.
5. Загружаются модули тестов в порядке, заданном переменной TEST_SUITES. Вывод команд, выполняемых через SSH доступен в файле ${WORKDIR}/testimgage/ssh_target_log.
6. При отсутствии отказов тестирование выполняется до завершения с записью вывода в файл ${WORKDIR}/temp/log.do_testimage.

3.29.1.2. Включение тестов при работе на аппаратной платформе

Система сборки OE может выполнять тесты на реальном оборудовании, а для некоторых устройств можно развернуть образ для предварительного тестирования на устройстве. Для автоматизированного развёртывания на устройстве устанавливается первичный (master) образ. Затем при каждом запуске теста выполняются перечисленные операции.

1. Загружается первичный образ, используемый для записи тестируемого образа во второй раздел.

2. Устройство перезагружается с помощью внешнего сценария, который нужно создать.
3. Загружается тестируемый образ.

При запуске тестов (независимо от способа разворачивания образа) устройство предполагается подключённым к сети с известным адресом IP, который может быть статическим или полученным от сервера DHCP. Для запуска тестов на устройстве нужно установить в переменной TEST_TARGET подходящее значение (для QEMU это не требовалось).

- *SimpleRemoteTarget* для запуска тестов на целевой системе, в которой уже имеется тестируемый образ и доступна сеть. Этот вариант пригоден для реального устройства или образа, запущенного в QEMU или на иной виртуальной машине.
- *SystemdbootTarget* для машин на основе EFI с загрузчиком systemd-boot и установленным core-image-testmaster (или аналогом). Устройство должно быть подключено к сети с поддержкой DHCP и сохранением IP-адреса при перезагрузке. Для этого режима имеются дополнительные требования, описанные в параграфе 3.29.1.3. Выбор SystemdbootTarget.
- *BeagleBoneTarget* при разворачивании образов и запуске тестов на устройстве BeagleBone "Black" или оригинальном "White". Описание тестов дано в файле meta-yocto-bsp/lib/oeqa/controllers/beaglebonetarget.py.
- *EdgeRouterTarget* при разворачивании образов и запуске тестов на устройстве Ubiquiti Networks EdgeRouter Lite. Описание тестов дано в файле meta-yocto-bsp/lib/oeqa/controllers/edgeroutertarget.py.
- *GrubTarget* при разворачивании образов и запуске тестов на ПК с загрузчиком GRUB. Описание тестов дано в файле meta-yocto-bsp/lib/oeqa/controllers/grubtarget.py.
- *your-target* при разворачивании образов и запуске тестов на машине с пользовательским уровнем BSP. Для этого нужно добавить модуль Python, определяющий класс цели, в каталог lib/oeqa/controllers/ на своём уровне, а также обеспечить пустой файл __init__.py. Примеры даны в meta-yocto-bsp/lib/oeqa/controllers/.

3.29.1.3. Выбор SystemdbootTarget

При выборе TEST_TARGET = "SystemdbootTarget" нужно выполнить однократную настройку первичного образа.

1. *Установка* EFI_PROVIDER = "systemd-boot".
2. *Сборка первичного образа* core-image-testmaster. Задание для образа приведено в качестве примера и может быть настроено в соответствии с задачей. Требования к образу приведены ниже.
   - Наследование core-image для установки модулей ядра.
   - Установка обычных утилит Linux, а не busybox (bash, coreutils, tar, gzip, kmod).
   - Применение образа initramfs с настроенным установщиком. Обычно образ создаёт один раздел rootfs, но здесь применяется образ, создающий определённый набор разделов. Установщик подходит не для всех BSP и может потребоваться создание разделов вручную по описанной ниже схеме.
     - Первый раздел с точкой монтирования /boot и меткой "boot".
     - Основной раздел rootfs для установки образа с точкой монтирования /.
     - раздел с меткой "testrootfs" для разворачивания тестируемого образа.
3. *Установка образа,* созданного для целевой системы.

В заключение нужно установить тестируемый образ.

1. *Настройка файла local.conf* путём включения приведённых ниже строк.

```
IMAGE_FSTYPES += "tar.gz"
INHERIT += "testimage"
TEST_TARGET = "SystemdbootTarget"
TEST_TARGET_IP = "192.168.2.3"
```

2. *Сборка тестируемого образа* с помощью команды bitbake core-image-sato.

3.29.1.4. Управление питанием

Для большинства аппаратных платформ (кроме SimpleRemoteTarget) возможно управление питанием.

- Можно использовать TEST_POWERCONTROL_CMD вместе с TEST_POWERCONTROL_EXTRA_ARGS как команду, выполняемую на хосте для управления питанием. Код теста передаёт один аргумент - off, on или cycle (off, затем on). Например, в файле local.conf можно задать TEST_POWERCONTROL_CMD = "powercontrol.exp test 10.11.12.1 nuc1". В этом случае предполагается, что сценарий выполняет команду ssh test@10.11.12.1 "pyctl nuc1 arg", затем выполняется сценарий Python для управления питанием с именем nuc1.

  Нужно настроить TEST_POWERCONTROL_CMD и TEST_POWERCONTROL_EXTRA_ARGS для своей ситуации. Единственным требованием является указание on, off или cycle в качестве последнего аргумента.
- Когда команда не задана, происходит подключение к устройству через SSH и выполняется обычная команда reboot для перезагрузки устройства. Это хорошо в тех случаях, когда машина действительно перезагружается (тест SSH не даёт отказа). Это полезно в простых ситуациях, обычно с одной платой, где время от времени возможно ручное воздействие.

Если оборудование не поддерживает управление питанием, но нужно провести эксперимент с автоматизированным тестированием устройства, можно использовать сценарий dialog-power-control, который выводит диалог, приглашающий выполнить нужную операцию управления питанием. Для этого сценария нужно установить KDialog или Zenity и задать TEST_POWERCONTROL_CMD = "${COREBASE}/scripts/contrib/dialog-power-control".

3.29.1.5. Подключение последовательной консоли

Если тест на целевой машине требует последовательную консоль для взаимодействия с загрузчиком (например, BeagleBoneTarget, EdgeRouterTarget, GrubTarget), нужно указать команду соединения с консолью на целевой машине в переменной TEST_SERIALCONTROL_CMD и возможно в TEST_SERIALCONTROL_EXTRA_ARGS.

Это может быть терминальная программа, если машина подключена к локальному последовательному порту, а также telnet или ssh при удалённом подключении к консоли. Независимо от способа программе нужно просто подключиться к последовательной консоли и перенаправить это соединение на стандартный ввод и вывод, как делает любая терминальная программа. Например, при использовании программы picocom на последовательном порту /dev/ttyUSB0 во скоростью 115200 бит/с можно задать TEST_SERIALCONTROL_CMD = "picocom /dev/ttyUSB0 -b 115200".

Для локальных устройств, где последовательное устройство исчезает при перезагрузке, предоставляется дополнительный сценарий serdevtry, который просто указывается в качестве префикса команды вызова терминала с указанием пути. Например, TEST_SERIALCONTROL_CMD = "${COREBASE}/scripts/contrib/serdevtry picocom -b 115200 /dev/ttyUSB0".

3.29.2. Запуск тестов

Тесты можно запускать автоматически или вручную.

- *Автоматический запуск.* Для автоматического запуска теста после создания образа системой сборки OE нужно сначала установить в файле local.conf каталога сборки TESTIMAGE_AUTO = "1", а затем собрать образ. После успешной сборки вводится команда bitbake core-image-sato.
- *Запуск вручную.* Для запуска вручную нужно сначала глобально унаследовать класс testimage, включив в файл local.conf строку INHERIT += "testimage", а затем запустить тесты командой bitbake -c testimage image.

Все файлы тестов хранятся в каталоге meta/lib/oeqa/runtime внутри дерева исходных кодов, имена тестов напрямую отображаются на модули Python. Каждый модуль может включать множество тестов, которые обычно группируются по функциональным областям (например, тесты для systemd собраны в модуле meta/lib/oeqa/runtime/systemd.py).

Тесты можно добавить на любой уровень, разместив их в подходящем месте (layer/lib/oeqa/runtime) и добавив его в переменную BBPATH файла local.conf file. Имена модулей не должны совпадать с именами из meta/lib/oeqa/runtime.

Можно изменить набор тестов путём добавления или переопределения переменной TEST_SUITES в local.conf, где каждое имя представляет требуемый для образа тест. Модули из TEST_SUITES не могут быть пропущены, даже когда тест не подходит для образа (например, запуск тестов RPM на образе без rpm). Добавление auto в TEST_SUITES заставляет систему сборки пытаться запустить все подходящие для образа тесты (т. е. каждый модуль может пропустить себя). Порядок тестов в TEST_SUITES важен и зависит от зависимостей между тестами, поэтому тест, зависящий от другого теста, нужно указывать после того, от которого он зависит. Например, поскольку тест ssh зависит от теста ping, следует помещать ssh в список после ping. Класс test не меняет порядок и не контролирует зависимости.

Каждый метод может иметь несколько классов с разными методами тестирования. Применяются правила Python unittest. Ниже указаны некоторые обстоятельства, которые нужно принимать во внимание при запуске тестов.

- По умолчанию тесты для образа определяется в виде DEFAULT_TEST_SUITES_pn-image = "ping ssh df connman syslog xorg scp vnc date rpm dnf dmesg".
- Добавление своих тестов в список имеет форму TEST_SUITES_append = " mytest".
- Конкретный набор тестов запускается в виде TEST_SUITES = "test1 test2 test3".

3.29.3. Экспорт тестов

Можно экспортировать тесты для запуска независимо от системы сборки. Экспорт требуется, если нужно запускать тесты вручную без планировщика. Можно экспортировать лишь тесты, указанные в TEST_SUITES.

Если образ уже собран, нужно проверить наличие в local.conf приведённых ниже строк.

```
INHERIT +="testexport"
TEST_TARGET_IP = "IP-address-for-the-test-target"
TEST_SERVER_IP = "IP-address-for-the-test-server"
```

После этого тесты можно экспортировать командой bitbake image -c testexport. Экспортируемые тесты помещаются в каталог tmp/testexport/image области сборки, указываемый переменной TEST_EXPORT_DIR.

Экспортированные тесты можно запускать извне среды сборки, как показано ниже.

```
$ cd tmp/testexport/image
$ ./runexported.py testdata.json
```

Ниже приведён пример, показывающий адреса IP и использующий образ core-image-sato:

```
INHERIT +="testexport"
TEST_TARGET_IP = "192.168.7.2"
TEST_SERVER_IP = "192.168.7.1"
```

Тест экспортируется командой bitbake core-image-sato -c testexport и запускается извне, как показано ниже.

```
$ cd tmp/testexport/core-image-sato
$ ./runexported.py testdata.json
```

3.29.4. Создание новых тестов

Все новые тесты нужно размещать в корректном месте, чтобы система сборки могла найти их. Тесты функций, добавляемых тем или иным уровнем, следует размещать в каталоге layer/lib/oeqa/runtime (при условии обычного расширения переменной BBPATH в файле уровня layer.conf). Следует помнить отмеченные ниже детали.

- Имена файлов должны напрямую отображаться на имена тестов (модулей).
- Не допускается использование имён, совпадающих с именами имеющихся тестов.

- Как минимум в каталоге запуска тестов должен быть пустой файл __init__.py.

Создание нового теста уместно начать с копирования имеющегося (например, syslog.py или gcc.py удобны в качестве образца). Модули тестов могут использовать код из вспомогательных классов meta/lib/oeqa/utils.

Команды оболочки следует структурировать так, чтобы на них можно было полагаться, а при успехе они бы возвращали один код. Следует учитывать необходимость анализа вывода в некоторых случаях. Примеры модулей можно найти в файлах df.py и date.py.

Все классы тестов наследуют oeRuntimeTest из файла meta/lib/oetest.py. Этот базовый класс предлагает некоторые атрибуты, описанные ниже.

3.29.4.1. Методы класса

- *hasPackage(pkg)* возвращает True, если пакет pkg имеется в списке установленных в образ, основанном на файле манифеста, создаваемом задачей do_rootfs task.
- *hasFeature(feature)* возвращает True если feature имеется в IMAGE_FEATURES или DISTRO_FEATURES.

3.29.4.2. Атрибуты класса

- *pscmd* эквивалентна "ps -ef", если procps имеется в образе. В остальных случаях pscmd будет "ps" (busybox).
- *tc* - контекст вызванного теста с доступом к перечисленным ниже атрибутам.
  - *d* - хранилище данных BitBake, позволяющие использовать такие вызовы, как oeRuntimeTest.tc.d.getVar("VIRTUAL-RUNTIME_init_manager").
  - *testslist u testsrequired* служат для внутреннего использования и не нужны тестам.
  - *filesdir* - абсолютный путь к meta/lib/oeqa/runtime/files, где содержатся вспомогательные файлы тестов для копирования в целевую систему (такие, как небольшие файлы C для компиляции).
  - *target* - объект контроллера целевой системы, используемый для развёртывания и запуска определённого образа (например, QemuTarget, SimpleRemote, SystemdbootTarget). Тесты обычно используют приведённые ниже элементы.
    - *ip* - IP-адрес целевой системы.
    - *server_ip* - IP-адрес хоста, который обычно используется тестами DNF.
    - *run(cmd, timeout=None)* - единственный, наиболее используемый метод. Команда служит «оболочкой» для ssh root@host "cmd" и возвращает пару (status, output) - код возврата cmd и её вывод. Необязательный аргумент timeout указывает число секунд, в течение которых тесту следует ожидать возврата cmd. Значение None задаёт принятый по умолчанию интервал 300 секунд, 0 задаёт неограниченное ожидание.
    - *copy_to(localpath, remotepath)* - scp localpath root@ip:remotepath.
    - *copy_from(remotepath, localpath)* - scp root@host:remotepath localpath.

3.29.4.3. Атрибуты экземпляра

Имеется один атрибут экземпляра - target, который идентичен атрибуту класса с тем же именем, описанному выше. Этот атрибут существует для экземпляра и класса, поэтому тесты могут использовать в методах экземпляра self.target.run(cmd) вместо oeRuntimeTest.tc.target.run(cmd).

3.29.5. Установка пакетов на тестируемом устройстве без менеджера пакетов

Когда для теста нужен пакет, созданный BitBake, этот пакет можно установить на тестируемом устройстве без менеджера пакетов. Однако потребуется соединение SSH, а целевая система должна использовать класс sshcontrol. Этот метод использует команду scp для копирования файлов между хостом и целью, что ведёт к потере прав доступа и специальных атрибутов.

Используется файл JSON для определения требуемых тесту пакетов. Этот файл должен размещаться там же, где находится определяющий тест файл. Кроме того, имя файла должно напрямую отображаться на имя тестового модуля и использовать расширение .json. Файл должен включать объект с именем теста в качестве ключа объекта или массива. Этот объект (массив объектов) использует указанные ниже данные.

- "pkg" - обязательная строка, указывающая имя устанавливаемого пакета.
- "rm" - необязательное логическое значение, управляющее удалением пакета после теста (по умолчанию false).
- "extract" - необязательное логическое значение, управляющее распаковкой формата пакета (по умолчанию false). При значении true пакет не устанавливается автоматически на тестируемом устройстве.

Далее приведён пример файла JSON для теста foo, устанавливающего пакет bar, и теста foobar, устанавливающего пакеты foo и bar. По завершении теста пакеты на устройстве удаляются.

```
    {
        "foo": {
"pkg": "bar"
        },
        "foobar": [
{
    "pkg": "foo",
    "rm": true
},
{
```

```
    "pkg": "bar",
    "rm": true
}
        ]
    }
```

3.30. Средства и методы отладки

Конкретный метод отладки зависит от природы проблемы и области, где возникает ошибка. Стандартные методы отладки на основе сравнение с последним работающим вариантом и проверки внесённых изменений и поэтапным их анализом для поиска источника проблемы применимы в YP как и других системах. Хоть и невозможно детализировать каждый отказ, в этом разделе даны некоторые базовые советы, которые помогут при устранении проблем.

Для отладки полезны инструменты отчётов об ошибках. Включение этих инструментов в YP заставляет систему сборки OE выводить сообщения об ошибках на консоль сборки. По завершении сборки можно сохранить эту информацию в общей базе данных, которая может помочь в поиске источников проблем. Информация о работе с инструментом информирования об ошибках приведена в параграфе 3.33. Использование инструмента отчётов об ошибках.

3.30.1. Просмотр журналов отказов задач

Журнал задачи можно найти в файле ${WORKDIR}/temp/log.do_taskname. Например, журнал задачи do_compile минимального образа QEMU для машины x86 (qemux86) будет в файле tmp/work/qemux86-poky-linux/core-image-minimal/1.0-r0/temp/log.do_compile. Для просмотра команд BitBake, связанных с журналом, следует обратиться к соответствующему файлу run.do_taskname в том же каталоге. Файлы log.do_taskname и run.do_taskname на деле являются символьными ссылками на файлы log.do_taskname.pid и log.run_taskname.pid, где pid является идентификатором PID выполнявшейся задачи. Символьный ссылки всегда указывают на последние файлы.

3.30.2. Просмотр значений переменных

Иногда нужно узнать значение переменной на этапе анализа BitBake. Это может быть связано с неожиданным поведением проекта. Причиной может быть, например, неудачная попытка изменить переменную. Опция BitBake -e позволяет увидеть значения переменных после анализа файлов конфигурации (local.conf, bblayers.conf, bitbake.conf и т. д.). Команда bitbake -e recipename выведет значения переменных после анализа указанного задания.

Каждое задание имеет свой набор переменных (хранилище данных). После анализа конфигурации делается копия хранилища до начала анализа заданий. Такое копирование предполагает, что набор переменных одного задания не виден другим заданиям. В задании каждая задача имеет доступ к хранилищу данных задания и установленные задачей переменные доступны другим задачам в рамках этого задания.

В выводе команды bitbake -e каждой переменной предшествует информация о способе её назначения, включая временные значения, которые были переопределены, и установленные флаги (varflags). Это удобно для отладки. Переменные, экспортируемые в среду, указываются с префиксом export в выводе bitbake -e. Например, export CC="i586-poky-linux-gcc -m32 -march=i586 —sysroot=/home/ulf/poky/build/tmp/sysroots/qemux86".

Кроме значений переменных по команде bitbake -e или bitbake -e recipe выводятся приведённые ниже данные.

- Вывод начинается с дерева, где указаны все файлы конфигурации и классы, включённые глобально, с рекурсивным указанием включаемых и наследуемых файлов. Большая часть поведения системы сборки OE (включая поведение обычных задач сборки задания) реализована в классе base и наследуемых им классах, а не в самой программе BitBake.
- После значений переменных выводятся все функции. Для функций оболочки включённые в них переменные преобразуются. Если функция изменяется путём переопределения или операторов со стилем переопределения, таких как _append и _prepend, выводится окончательная функция.

3.30.3. Просмотр данных пакетов с помощью oe-pkgdata-util

Утилита oe-pkgdata-util позволяет запрашивать PKGDATA_DIR и выводит связанную с пакетом информацию для пакетов, которые уже собраны. Ниже описаны некоторые субкоманды oe-pkgdata-util. В именах пакетов и путях допускается использование обычных шаблонов * и ?.

- *oe-pkgdata-util list-pkgs [pattern]* выводит данные о всех или соответствующих шаблону pattern пакетах.
- *oe-pkgdata-util list-pkg-files package ...* выводит файлы и каталоги указанных пакетов. Другим вариантом просмотра содержимого пакетов является изучение каталога ${WORKDIR}/packages-split в задании для пакета. Этот каталог создаётся задачей do_package и включает один подкаталог для каждого создаваемого заданием пакета. Для просмотра каталогов ${WORKDIR}/packages-split нужно отключить rm_work при сборке задания.
- *oe-pkgdata-util find-path path ...* выводит имена всех пакетов, включающих указанные пути. Например, приведённая ниже команда говорит, что файл /usr/share/man/man1/make.1 включён в пакет make-doc.

```
$ oe-pkgdata-util find-path /usr/share/man/man1/make.1
make-doc: /usr/share/man/man1/make.1
```

- *oe-pkgdata-util lookup-recipe package ...* выводит имена заданий, создавших указанные пакеты.

Для получения дополнительной информации о командах oe-pkgdata-util можно использовать команды

```
$ oe-pkgdata-util --help
$ oe-pkgdata-util subcommand --help
```

3.30.4. Просмотр зависимостей между заданиями и задачами

Иногда бывает сложно понять, почему BitBake хочет собрать другие задания перед сборкой указанного. Информация о зависимостях помогает в этом. Для получения данных о зависимостях задания служит команда bitbake -g recipename, которая выводит в текущий каталог два файла, указанных ниже.

- *pn-buildlist* - список задач и целей, вовлечённых в сборку задания recipename. Вовлеченность означает, что по меньшей мере одна из задач задания нужна при сборке recipename с нуля. Цели, указанные в ASSUME_PROVIDED, не выводятся.

- *task-depends.dot* - граф зависимостей между задачами.

Граф выводится в формате DOT и может быть преобразован в изображение (например, с помощью dot из Graphviz).

- Файлы DOT имеют текстовый формат. Граф создаётся с помощью команды bitbake -g и часто бывает достаточно большим для чтения без специальной очистки (например, опции Bitbake -l) и обработки. Тем не менее, файлы .dot могут быть полезны для получения информации. Например, строка "libxslt.do_configure" -> "libxml2.do_populate_sysroot" в файле task-depends.dot указывает, что задача do_configure в libxslt зависит от задачи do_populate_sysroot в libxml2, которая указана в DEPENDS.

- Примером обработки файлов .dot является сценарий Python scripts/contrib/graph-tool, который находит и показывает пути между узлами графа.

Другим вариантом просмотра данных о зависимостях служит команда bitbake -g -u taskexp recipename, которая выводит окно графического интерфейса с зависимостями при сборке и работе для заданий, включённых в сборку recipename.

3.30.5. Просмотр зависимостей между переменными задач

Как отмечено в разделе Checksums (Signatures) [6], BitBake пытается автоматически определить, от каких переменных зависит задача, если значения переменных меняются. Это определение обычно надёжно, однако в таких ситуациях, как создание имён переменных в процессе работы, может потребоваться вручную указывать зависимости от этих переменных с помощью vardeps, как описано в разделе Variable Flags [6]. Если нет уверенности в автоматическом определении зависимостей переменной для данной задачи, можно посмотреть зависимости, найденные BitBake.

1. Сборка задания, содержащего задачу по команде bitbake recipename.

2. Просмотр в каталоге STAMPS_DIR файла данных подписи (sigdata), соответствующего задаче. Файлы sigdata содержат базу данных Python со всеми метаданными, использованными при создании контрольной суммы для задачи. Например, для do_fetch в задании db файл sigdata можно найти в каталоге ${BUILDDIR}/tmp/stamps/i586-poky-linux/db/6.0.30-r1.do_fetch.sigdata.7c048c18222b16ff0bcee2000ef648b1. Для задач, ускоренных через кэш общего состояния (sstate), создаётся дополнительный файл siginfo в SSTATE_DIR вместе с кэшированным выводом задачи. Файлы siginfo содержат те же данные, что и sigdata.

3. Ввод команды bitbake-dumpsig для файла sigdata или siginfo. Например, bitbake-dumpsig ${BUILDDIR}/tmp/stamps/i586-poky-linux/db/6.0.30-r1.do_fetch.sigdata.7c048c18222b16ff0bcee2000ef648b1. Вывод этой команды позволяет увидеть строки предполагаемых зависимостей переменных для задачи с учётом рекурсивных зависимостей. Например, Task dependencies: ['PV', 'SRCREV', 'SRC_URI', 'SRC_URI[md5sum]', 'SRC_URI[sha256sum]', 'base_do_fetch']. Функции (например, base_do_fetch) также учитываются в зависимостях переменных и сами зависят от указанных в них переменных. Вывод bitbake-dumpsig включает также значение каждой переменной, список её зависимостей, и данные BB_HASHBASE_WHITELIST.

Имеется также команда bitbake-diffsigs для сравнения двух файлов siginfo или sigdata, которая может быть полезна при поиске различий между двумя версиями задачи. При вызове команды для одного файла она ведёт себя как bitbake-dumpsig. Можно также использовать BitBake для вывода данных подписи без выполнения задачи с помощью опции BitBake --dump-signatures=SIGNATURE_HANDLER или -S SIGNATURE_HANDLER. Основными значениями SIGNATURE_HANDLER являются none и printdiff, обеспечивающие вывод лишь подписи или сравнение подписи с кэшированной. Использование BitBake с любой из этих опций заставляет BitBake вывести дамп файлов sigdata в каталоге stamps для каждой задачи, которая будет выполняться, без реальной сборки указанного пакета.

3.30.6. Просмотр метаданных, использованных для создания входной подписи

Просмотр метаданных, используемых для создания входной подписи sstate также может помочь при отладке. Эта информация доступна в файлах siginfo каталога SSTATE_DIR. Интерпретация данных описана в предыдущем параграфе. Концепции общего состояния рассмотрены в разделе Shared State [1].

3.30.7. Аннулирование общего состояния для повторного запуска задачи

Система сборки OE использует контрольные суммы и кэш общего состояния для предотвращения ненужного повтора сборки задач. Эту схему называют кодом общего состояния и как все схемы, она обладает некоторыми недостатками. Возможны случаи неявного внесения в код изменений, которые не будут учтены при расчёте контрольной суммы. Эти изменения влияют на вывод задачи, но не вызывают код общего состояния в повторной сборке задания. Рассмотрим пример, где инструмент меняет вывод rpmdeps. Результатом изменения должна стать недействительность всех элементов кэша общего состояния для package и package_write_rpm. Однако изменение является внешним (неявным), поэтому записи кэша общего состояния остаются корректными и процесс сборки будет использовать их вместо перезапуска задачи, что может вызвать проблемы.

Для предотвращения проблем при сборке нужно понимать последствия вносимых изменений, помня о том, что вносимые напрямую в функцию изменения отражаются в контрольной сумме, делая соответствующую область кэша общего состояния недействительной. Однако следует помнить, что неявные изменения не очевидны и могут влиять на результаты выполнения задачи. При обнаружении неявных изменений можно предпринять действия по аннулированию кэша для повторного запуска задачи. Например, для аннулирования файлов общего состояния пакета можно изменить комментарии задачи do_package или какой-либо из вызываемых ею функций. Хотя это изменение будет косметическим, оно повлияет на контрольную сумму и заставит систему сборки заново выполнить задачу. Пример такого подхода доступен по ссылке.

3.30.8. Запуск конкретных задач

Любое задание состоит из набора задач. Стандартный порядок выполнения задач в BitBake имеет вид - do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install, do_package, do_package_write_*, do_build. По умолчанию

выполняется задача do_build и все задачи, которые требуется выполнить до неё. Некоторые задачи (например, do_devshell) не входят по умолчанию в цепочку сборки. Для запуска таких задач можно использовать опцию -c в BitBake, например,

```
$ bitbake matchbox-desktop -c devshell
```

Опция -c учитывает зависимости задач и будет заранее запускать все задачи, от которых зависит выполняемая задача (включая задачи из других заданий). Даже при указании задачи вручную с помощью опции -c программа будет запускать лишь те задачи, которые она сочтёт нужными. Выбор актуальных задач описан в разделе Stamp Files and the Rerunning of Tasks [1].

Если нужно форсировать запуск устаревшей задачи (например, после внесения вручную изменений в каталог WORKDIR задания), можно воспользоваться опцией -f. Эта опция никогда не требуется при работе с задачей do_devshell, поскольку для этой задачи уже установлен флаг переменной [nostamp]. Ниже приведён пример использования опции -f.

```
$ bitbake matchbox-desktop
...
Внесение изменений в рабочий каталог
...
$ bitbake matchbox-desktop -c compile -f
$ bitbake matchbox-desktop
```

Эта последовательность сначала собирает, а затем заново компилирует matchbox-desktop. Последняя команда перезапускает все задачи (в основном, упаковку) поле компиляции. BitBake понимает, что задача do_compile была запущена повторно и это требует перезапуска других задач.

Более короткий способ перезапуска задачи и всех обычных задач сборки задания, зависящих от неё., обеспечивает опция -C (не следует путать с -c). Опция аннулирует данную задачу и запускает задачу do_build, которая используется по умолчанию, и задачи от которых та зависит. Для этого две последних команды в приведённом выше примере следует заменить командой bitbake matchbox-desktop -C compile. Опции -f и -C работают за счёт изменения входной контрольной суммы указанной задачи, что косвенно вызывает перезапуск задачи и зависимых от неё. задач с использованием обычных механизмов зависимости.

BitBake явно отслеживает изменение задач таким способом и будет выдавать при следующей сборке, включающей такие задачи, предупреждение вида WARNING: /home/ulf/poky/meta/recipes-sato/matchbox-desktop/matchbox-desktop_2.1.bb.do_compile is tainted from a forced run. Цель этого предупреждения заключается в уведомлении пользователя о том, что рабочий каталог и вывод сборки могут не быть «чистыми» как при нормальной сборке. Чтобы избежать таких предупреждений, можно очистить рабочий каталог и заново собрать задание, как показано ниже.

```
$ bitbake matchbox-desktop -c clean
$ bitbake matchbox-desktop
```

Список задач данного пакета можно посмотреть, запустив задачу do_listtasks командой вида bitbake matchbox-desktop -c listtasks. Результат будет выведен на консоль и записан в файл ${WORKDIR}/temp/log.do_listtasks.

3.30.9. Отладочный вывод BitBake

Отладочный вывод BitBake обеспечивает опция -D, которую можно указывать до 3 раз (-DDD) для расширения подробностей. Команда bitbake -DDD -v targetname может показать, почему программа BitBake выбрала ту или иную версию пакета и провайдера, а также помогает в ситуациях, когда результат представляется неожиданным.

3.30.10. Сборка без зависимостей

Для сборки конкретного задания (файл.bb) можно применить команду вида bitbake -b somepath/somerecipe.bb. Команда не проверяет зависимостей, поэтому нужно проверить их выполнение. Имя файла можно указать частично и BitBake будет искать уникальное совпадение.

3.30.11. Механизмы протоколирования заданий

YP обеспечивает несколько механизмов протоколирования для отладочного вывода и уведомлений об ошибках и предупреждениях. Ниже перечислены функции Python для протоколирования, обеспечивающие запись в файлы журналов ${T}/log.do_task и на стандартный вывод.

- *bb.plain(msg)* записывает msg в журнальный файл и на стандартный вывод.
- *bb.note(msg)* записывает "NOTE: msg" в журнальный файл, а также на стандартный вывод (с опцией -v).
- *bb.debug(level, msg)* записывает "DEBUG: msg" в журнальный файл, а также на стандартный вывод, если уровень не меньше значения level (см. опцию -D [6]).
- *bb.warn(msg)* записывает "WARNING: msg" в журнальный файл и на стандартный вывод.
- *bb.error(msg)* записывает "ERROR: msg" в журнальный файл и на стандартный вывод. Вызов этой функции не ведёт к отказу задачи.
- bb.fatal(msg) похожа на bb.error(msg), но ведёт к отказу задачи.

  bb.fatal() создаёт исключительную ситуацию, означающую, что не нужно помещать return после функции.

Такие же функции протоколирования доступны для функций оболочки bbplain, bbnote, bbdebug, bbwarn, bberror, bbfatal) и реализованы в классе logging (см. каталог meta/classes в дереве исходных кодов).

3.30.11.1. Журналы Python

При подготовке заданий, использующих Python, с кодом обслуживания журналов сборки следует учитывать, что журналы предназначены в основном для информирования с минимизацией вывода на консоль. Если нужны сообщения о состоянии, следует указывать уровень протоколирования debug.

Ниже приведён пример на языке Python для обработки журналов в плане определения числа задач, которые нужно запустить. Дополнительная информация приведена в разделе do_listtasks [3]).

```
    python do_listtasks() {
        bb.debug(2, "Starting to figure out the task list")
        if noteworthy_condition:
bb.note("There are 47 tasks to run")
        bb.debug(2, "Got to point xyz")
        if warning_trigger:
bb.warn("Detected warning_trigger, this might be a problem later.")
        if recoverable_error:
bb.error("Hit recoverable_error, you really need to fix this!")
        if fatal_error:
bb.fatal("fatal_error detected, unable to print the task list")
        bb.plain("The tasks present are abc")
        bb.debug(2, "Finished figuring out the tasklist")
    }
```

3.30.11.2. Журналы Bash

При подготовке заданий, использующих Bash, с кодом обслуживания журналов сборки следует учитывать, что журналы предназначены в основном для информирования с минимизацией вывода на консоль. Синтаксис, применяемый для заданий, написанных на Bash, аналогичен синтаксису заданий на Python, описанных в предыдущем параграфе. Ниже приведён пример для Bash, регистрирующий выполнение функции do_my_function.

```
    do_my_function() {
        bbdebug 2 "Running do_my_function"
        if [ exceptional_condition ]; then
bbnote "Hit exceptional_condition"
        fi
        bbdebug 2  "Got to point xyz"
        if [ warning_trigger ]; then
bbwarn "Detected warning_trigger, this might cause a problem later."
        fi
        if [ recoverable_error ]; then
bberror "Hit recoverable_error, correcting"
        fi
        if [ fatal_error ]; then
bbfatal "fatal_error detected"
        fi
        bbdebug 2 "Completed do_my_function"
    }
```

3.30.12. Отладка конфликтов параллельной сборки

Параллельная сборка представляет собой одновременное выполнение нескольких частей задания, при котором могут возникать ситуации, когда требуемые для работы результаты ещё не готовы. Такие конфликты раздражают, а порой их сложно воспроизвести и устранить. Однако есть ряд простых рекомендаций, которые помогают решить проблему. Ниже представлен реальный пример ошибки, с которой сталкивается YP autobuilder и способ её исправления.

Если конфликт не удаётся исправить, можно попытаться сбросить PARALLEL_MAKE или PARALLEL_MAKEINST.

3.30.12.1. Отказы

Предположим, например, что собирается образ, который зависит от пакета neard, и при сборке BitBake сталкивается с проблемой, выводя показанные ниже сообщения (строки вывода дополнительно разбиты для удобства восприятия).

```
| DEBUG: SITE files ['endian-little', 'bit-32', 'ix86-common', 'common-linux', 'common-glibc', 'i586-linux', 'common']
| DEBUG: Executing shell function do_compile
| NOTE: make -j 16
| make --no-print-directory all-am
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/types.h include/near/types.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/log.h include/near/log.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/plugin.h include/near/plugin.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/tag.h include/near/tag.h
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/adapter.h include/near/adapter.h
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/ndef.h include/near/ndef.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/tlv.h include/near/tlv.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/setting.h include/near/setting.h
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| /bin/mkdir -p include/near
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/device.h include/near/device.h
```

```
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/nfc_copy.h include/near/nfc_copy.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/snep.h include/near/snep.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/version.h include/near/version.h
| ln -s /home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/work/i586-poky-linux/neard/
  0.14-r0/neard-0.14/include/dbus.h include/near/dbus.h
| ./src/genbuiltin nfctype1 nfctype2 nfctype3 nfctype4 p2p > src/builtin.h
| i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/
  build/build/tmp/sysroots/qemux86 -DHAVE_CONFIG_H -I. -I./include -I./src -I./gdbus  -I/home/pokybuild/
  yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/sysroots/qemux86/usr/include/glib-2.0
  -I/home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/tmp/sysroots/qemux86/usr/
  lib/glib-2.0/include  -I/home/pokybuild/yocto-autobuilder/yocto-slave/nightly-x86/build/build/
  tmp/sysroots/qemux86/usr/include/dbus-1.0 -I/home/pokybuild/yocto-autobuilder/yocto-slave/
  nightly-x86/build/build/tmp/sysroots/qemux86/usr/lib/dbus-1.0/include  -I/home/pokybuild/yocto-autobuilder/
  yocto-slave/nightly-x86/build/build/tmp/sysroots/qemux86/usr/include/libnl3
  -DNEAR_PLUGIN_BUILTIN -DPLUGINDIR=\""/usr/lib/near/plugins"\"
  -DCONFIGDIR=\""/etc/neard\"" -O2 -pipe -g -feliminate-unused-debug-types -c
  -o tools/snep-send.o tools/snep-send.c
| In file included from tools/snep-send.c:16:0:
| tools/../src/near.h:41:23: fatal error: near/dbus.h: No such file or directory
|  #include <near/dbus.h>
|                       ^
| compilation terminated.
| make[1]: *** [tools/snep-send.o] Error 1
| make[1]: *** Waiting for unfinished jobs....
| make: *** [all] Error 2
| ERROR: oe_runmake failed
```

3.30.12.2. Воспроизведение ошибок

Конфликты при параллельной сборке могут не возникать каждый раз, поэтому нужен способ воспроизведения таких ошибок. В приведённом ниже примере компиляция пакета neard вызывает проблему. Поэтому сначала нужно собрать neard локально. Перед началом сборки в переменной PARALLEL_MAKE файла local.conf устанавливается большое значение (например, "-j 20"), что повысит вероятность возникновения ошибки. Далее выполняется сборка по команде bitbake neard, а по её завершении запускается сборка командой bitbake neard -c devshell (см. раздел 3.8. Использование среды devshell). В среде devshell вводятся команды

```
$ make clean
$ make tools/snep-send.o
```

Это позволит чётко увидеть отказ. В данном случае отсутствует зависимость neard для цели Makefile, как можно видеть из сокращённого вывода, представленного ниже.

```
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/home/scott-lenovo/......
   ...
tools/snep-send.c
In file included from tools/snep-send.c:16:0:
tools/../src/near.h:41:23: fatal error: near/dbus.h: No such file or directory
 #include <near/dbus.h>
       ^
compilation terminated.
make: *** [tools/snep-send.o] Error 1
$
```

3.30.12.3. Подготовка исправлений

Поскольку отсутствует зависимость для цели Makefile, нужно исправить файл Makefile.am, создаваемый из Makefile.in. Можно использовать Quilt для создания patch-файла (см. раздел 3.7. Использование Quilt).

```
$ quilt new parallelmake.patch
Patch patches/parallelmake.patch is now on top
$ quilt add Makefile.am
File Makefile.am added to patch patches/parallelmake.patch
```

Здесь нужно отредактировать файл Makefile.am для добавления нужной зависимости. Например, это может быть строка вида

```
tools/snep-send.$(OBJEXT): include/near/dbus.h
```

После редактирования файла нужно создать patch-файл с помощью команды

```
$ quilt refresh
Refreshed patch patches/parallelmake.patch
```

Полученный файл исправления нужно добавить в каталог задания командой вида

```
$ cp patches/parallelmake.patch poky/meta/recipes-connectivity/neard/neard
```

В заключение нужно применить правки к сборке задания neard (neard-0.14.bb), чтобы оператор SRC_URI включал patch-файл. Переменная в результате будет иметь вид

```
SRC_URI = "${KERNELORG_MIRROR}/linux/network/nfc/${BPN}-${PV}.tar.xz \
  file://neard.in \
  file://neard.service.in \
  file://parallelmake.patch \
"
```

После этого можно завершить работу с devshell командой exit.

3.30.12.4. Тестирование сборки

После внесений всех исправления можно повторить локальную сборку командой bitbake neard и результат должен быть положительным. Затем можно снова открыть devshell и повторить очистку и сборку, как показано ниже.

```
$ bitbake neard -c devshell
$ make clean
$ make tools/snep-send.o
```

Проблем возникать не должно. После внесения всех корректировок их следует зафиксировать для задания в OE-Core и восходящем репозитории, чтобы проблема не повторялась у других (см. параграф 3.31.2. Представление изменений в YP).

3.30.13. Удалённая отладка с помощью GDB

GDB[1] позволяет проверять запущенные программы с целью поиска и исправления неполадок, а также выполнять анализ данных после отказа программ. GDB доступен в составе YP и по умолчанию устанавливается в образах SDK, описание которых приведено в разделе Images [3]. Описание GDB доступно на странице http://sourceware.org/gdb/. Для более эффективной работы следует установить отладочные (-dbg) пакеты для приложений, которые планируется отлаживать. Это сделает доступными отладочные символы и сделает вывод более информативным.

Иногда нехватка памяти или дискового пространства не позволяет использовать GDB на целевой платформе напрямую. GDB для работы нужно загрузить отладочную информацию и двоичные файлы, кроме того программе приходиться выполнять много расчётов для поиска информации (имена и значения переменных, трассировка стека и т. п.). Эти издержки повышают нагрузку на устройство и могут менять характеристики отлаживаемых программ. Для решения проблемы можно использовать на целевой платформе программу gdbserver, которая не загружает отладочную информацию. Вместо этого отладочную информацию обрабатывает экземпляр GDB на хосте отладки. Этот хост передаёт команды управления программе gdbserver для запуска и остановки отлаживаемой программы, а считывания и записи информации в память отлаживаемой программы. Вся отладочная информация загружается и обрабатывается на хосте GDB, что позволяет сервер gdbserver быть компактным и быстрым.

Поскольку хост GDB отвечает за загрузку и обработку отладочной информации, для отладки нужно убедиться, что этот хост имеет доступ к двоичным файлам с отладочными символами, а также обеспечить компиляцию для целевой платформы без оптимизации. Хост GDB также должен иметь локальный доступ ко всем библиотекам, используемым отладочной программой. Поскольку для gdbserver отладочная информация локально не нужна, её можно исключить из двоичных файлов для целевой системы. Однако для соответствия с двоичными файлами на хосте отладки компиляция должна выполняться без оптимизации. В соответствии с документацией GDB будем называть двоичный файл на целевой системе «подчиненным» (inferior). Ниже описан процесс удалённой отладки с использованием GDB.

1. *Настройка системы сборки для создания отладочной файловой системы.* В файле local.conf следует указать

   ```
   IMAGE_GEN_DEBUGFS = "1"
   IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
   ```

   Эти опции заставят систему сборки OE создать специальную файловую систему для отладки, которая будет включать соответствующий исходный код и символы отладки для развёртываемой файловой системы. Система сборки делает это путём просмотра содержимого разворачиваемой файловой системы и извлечения соответствующих пакетов -dbg. Отладочная файловая система не является полной и содержит лишь компоненты отладки. Она должна применяться вместе с полной файловой системой, как показано ниже.

2. *Настройка системы для включения gdbserver на удалённой файловой системе.* В файл local.conf или задание для образа нужно включить строку IMAGE_INSTALL_append = “ gdbserver", которая обеспечит сервер.

3. *Сборка среды* для создания образа и сопровождающей отладочной файловой системы командой bitbake image. Сборку кросс-компоненты GDB и её подготовку для отладки лучше всего выполнить путём сборки SDK с помощью команды bitbake -c populate_sdk image.

   Другим вариантом является сборка минимального инструментария, соответствующего целевой системе. Этот вариант более компактный и реализуется командой bitbake meta-toolchain.

   Ещё один метод заключается в автономной сборке GDB по команде bitbake gdb-cross-architecture. Это создаёт временную копию cross-gdb, которую можно использовать для отладки. Этот решение наиболее быстрое, но два предыдущих более эффективны при продолжительном использовании отладчика. При запуске gdb-cross, система сборки OE предложит реальный образ (например, gdb-cross-i586).

4. *Установка debugfs,* как показано ниже.

   ```
   $ mkdir debugfs
   $ cd debugfs
   $ tar xvfj build-dir/tmp-glibc/deploy/images/machine/image.rootfs.tar.bz2
   $ tar xvfj build-dir/tmp-glibc/deploy/images/machine/image-dbg.rootfs.tar.bz2
   ```

5. *Установка GDB.* Установите SDK (в случае его использования) и выполните сценарий настройки среды. Если применялась система сборки, GDB будет в каталоге build-dir/tmp/sysroots/host/usr/bin/architecture/architecture-gdb

6. *Загрузка целевой системы.* В случае использования QEMU следует прочесть документацию QEMU. Проверьте доступ хоста к целевой системе по протоколу TCP.

7. *Отладка программы* включает запуск gdbserver на целевой системе и GDB на хосте отладки. В приведённом ниже примере отлаживается пакет gzip

   ```
   root@qemux86:~# gdbserver localhost:1234 /bin/gzip –help
   ```

   Опции gdbserver описаны в документации GDB Server. После запуска gdbserver нужно запустить GDB на хосте отладки и настроить для отладчика соединение с целевой системой, как показано ниже.

   ```
   $ cd directory-holding-the-debugfs-directory
   $ arch-gdb

   (gdb) set sysroot debugfs
   (gdb) set substitute-path /usr/src/debug debugfs/usr/src/debug
   (gdb) target remote IP-of-target:1234
   ```

[1]GNU Project Debugger - отладчик проекта GNU.

После этого все остальное должно загружаться автоматически (двоичные файлы, символы и заголовки). Команда GDB set в примере может быть включена в пользовательский файл ~/.gdbinit и при запуске GDB будут выполнены остальные команды из файла.

8. *Развёртывание без пересборки образа.* Во многих случаях при отладке может потребоваться быстрое развёртывание нового двоичного файла в целевой системе без пересборки образа целиком. Одним из вариантов решения этой задачи является просто сборка нужной компоненты и копирование файлов непосредственно в debugfs целевой системы и хоста отладки. Например,

```
$ bitbake bash
$ bitbake -c devshell bash
$ cd ..
$ scp packages-split/bash/bin/bash target:/bin/bash
$ cp -a packages-split/bash-dbg/* path/debugfs
```

3.30.14. Отладка на целевой платформе с помощью GDB

В предыдущем параграфе описана удалённая отладка с помощью GDB, которая применяется чаще по причине аппаратных ограничений встраиваемых систем. Однако возможна отладка и непосредственно на целевой платформе для более мощных устройств, описанная ниже. Для реализации такой отладки нужно выполнить ряд действий.

1. Установить GDB на целевой платформе, путём добавления в конфигурацию строки IMAGE_INSTALL_append = " gdb" или IMAGE_FEATURES_append = " tools-debug".
2. Обеспечить наличие отладочных символов, например, с помощью строки IMAGE_INSTALL_append = " packagename-dbg" или IMAGE_FEATURES_append = " dbg-pkgs".

Для повышения точности отладочной информации можно снизить уровень оптимизации, используемый компилятором. Например, можно добавить в файл local.conf строку DEBUG_BUILD = "1", что снизит уровень оптимизации с FULL_OPTIMIZATION = "-O2" до DEBUG_OPTIMIZATION = "-O -fno-omit-frame-pointer". Это одновременно снизит производительность приложений, поэтому по завершении отладки следует восстановить оптимизацию.

3.30.15. Рекомендации по отладке

При добавлении пакетов нужно отслеживать появление нежелательных элементов в командах компилятора. Например, это могут быть ссылки на локальные файловые системы, такие как /usr/lib/ или /usr/include/. Если нужно исключить при загрузке заставку psplash, следует добавить psplash=false в командную строку ядра. Это позволит видеть консольный вывод. Можно также переключить виртуальную консоль (например, Fn+-> или Fn+<- на Zaurus).

Удаление TMPDIR (обычно tmp/ в каталоге сборки) часто решает временные проблемы сборки. Это не влечёт значительных издержек, поскольку вывод задач кэшируется в SSTATE_DIR (обычно sstate-cache/ в каталоге сборки). Однако это может быть лишь обходом, а не решением проблемы. Поэтому неплохо поискать решение до удаления каталога.

Понимание практического применения свойства (функции) в задании очень важно, поэтому рекомендуется настроить тот или иной метод поиска в файлах. Например, можно использовать приведённую ниже shell-функцию на основе GNU Grep, которая выполняет рекурсивный поиск теста в связанных с заданиями файлах, пропуская двоичные файлы, каталоги .git и каталог сборки (в предположении, что его имя начинается с build).

```
   g() {
       grep -Ir \
--exclude-dir=.git \
--exclude-dir='build*' \
--include='*.bb*' \
--include='*.inc*' \
--include='*.conf*' \
--include='*.py*' \
"$@"
   }
```

Ниже приведено несколько примеров использования функции.

```
$ g FOO    # рекурсивный поиск FOO
$ g -i foo # рекурсивный поиск foo без учёта регистра
$ g -w FOO # рекурсивный поиск FOO как слова с игнорированием FOOBAR
```

Если поиск информации о работе функции требует слишком много времени, это может говорить о необходимости расширить или улучшить документацию. В таких случаях уместно сообщить об ошибке с использованием YP Bugzilla. Работа с этим ресурсом описана на странице YP Bugzilla wiki и в параграфе 3.31.1. Фиксация ошибок в YP. В руководствах может не быть описания переменных, которые являются сугубо внутренними и имеют ограниченную область действия (например, переменные, используемые внутри одного файла .bbclass).

3.31. Внесение изменений в YP

Поскольку YP является открытым проектом, разрабатываемым сообществом, каждый может предложить в проект свои изменения или дополнения. В этом разделе описаны процедуры информирования о дефектах и внесения правок.

3.31.1. Фиксация ошибок в YP

Для сообщений о дефектах (ошибках) YP служит ресурс YP Bugzilla, информацию о котором можно найти в разделе Yocto Project Bugzilla [3]. Описание работы с ресурсом имеется на странице YP Bugzilla wiki. Ниже кратко описаны этапы информирования об ошибках.

1. Открыть страницу YP Bugzilla.
2. Выбрать ссылку File a Bug для ввода информации об ошибке.

3. Выбрать подходящие варианты Classification, Product и Component для информации об ошибке. Ошибки YP делятся на несколько категорий, включающих разную продукцию и компоненты. Например, для ошибки на уровне meta-intel следует выбрать Build System, Metadata & Runtime, BSPs и bsps-meta-intel.
4. Выбрать Version для YP в соответствии с версией, где найдена ошибка (например, 2.7.1).
5. Определить и выбрать важность (Severity) ошибки.
6. Выбрать оборудование (Hardware), с которым связана ошибка.
7. Выбрать архитектуру (Architecture), с которой связана ошибка.
8. Выбрать пункт изменения документации (Documentation change) для ошибки. Если влияние ошибки на документацию не понятно, следует выбрать Don't Know.
9. Представить краткое описание (Summary) ошибки в одну или две строки.
10. Представить подробное описание ошибки (Description), указав детали контекста, поведение, вывод и т. п. Здесь можно присоединить файлы с информацией, используя кнопку Add an attachment.
11. Нажать кнопку Submit Bug для фиксации сообщения, которому будет присвоен номер Bugzilla, а информация будет сохранена в системе отслеживания ошибок.

Полученные сообщения обрабатывает команда YP Bug Triage Team, присваивая ему дополнительные атрибуты (например, приоритет). Вы будете считаться «подателем» (Submitter) сообщения при всех последующих контактах. Bugzilla автоматически будет уведомлять по электронной почте о всех событиях, связанных с обработкой сообщения.

3.31.2. Представление изменений в YP

Приветствуется вклад в YP и OE. Поскольку система является настраиваемой и гибкой, очевидно, что разработчики захотят настроить и оптимизировать её для своих задач.

YP использует списки рассылки и рабочий процесс на основе исправления (patch), похожие на процессы для ядра Linux, но с существенными отличиями. Имеются специальные списки рассылки для представления правок. Сообщения из этих списков просматриваются и обрабатываются сопровождающими и нужно выбрать список в соответствии с размещением кода, который вы хотите изменить. Каждая компонента (например, уровень) должна включать файл README, указывающий, куда направить изменения и как их следует обрабатывать.

Можно отправить исправления в список рассылки, используя удобный способ создания patch-файлов. После отправки исправлений они обычно рассматриваются сообществом в целом и при обнаружении каких-либо проблем они указываются. Если исправления не создают негативных последствий, сопровождающий обычно принимает исправление, проверяет его и после успешного тестирования добавляет в код.

Репозиторий poky, являющийся эталонной средой сборки YP, является гибридным и состоит из нескольких частей (BitBake, Metadata, документация и т. п.), созданных с помощью инструмента combo-layer. Отправка изменений зависит от компонент, как указано ниже.

- *Core Metadata.* Отправляйте правки в список рассылки openembedded-core. Например, в этот список следует отправлять исправления для каталогов meta или scripts.
- *BitBake.* Отправляйте правки в список рассылки bitbake-devel.
- *meta-*.* Отправляйте правки в список рассылки poky.

Для изменений на других уровнях репозитория YP (yoctoproject.org), в инструментах и документации YP следует направлять правки в рассылку YP. Иногда конкретный список указан в документации уровня и следует применять его.

Для дополнительных заданий, не вписывающихся в основные метаданные, следует определить уровень, к которому относится задание, и представить правки в соответствии с документацией этого уровня (например, README). При наличии сомнений можно задать вопрос в общей рассылке Yocto или openembedded-devel.

Можно также отправить изменения в восходящий репозиторий и попросить сопровождающего внести их. Эти процедуры описаны в разделе Git Workflows and the Yocto Project [1].

В OpenEmbedded-Core имеется два тестовых репозитория:

- *ветвь "ross/mut"* - дерево mut (master-under-test) в репозитории poky-contrib источников YP;
- *ветвь "master-next" branch* является частью основного репозитория poky в репозитории источников YP.

Сопровождающие используют эти ветви для тестирования предложений перед включением правок. Включение правки в одну из этих ветвей позволяет получить представление о состоянии предложенных правок.

Эта система несовершенна и изменения могут иногда теряться в общем потоке. Запрос о статусе исправления или изменения является вполне уместным, если на предложение не было отклика. В YP планируется использование Patchwork для отслеживания статуса правок и автоматического предварительного просмотра.

3.31.2.1. Использование сценариев для представления изменений

Описанная здесь процедура служит для внесения изменений в восходящий репозиторий Git "contrib". Базовая информация о работе с репозиториями представлена в Git Community Book.

1. *Внесите изменения локально* в свой репозиторий Git. Изменения следует делать компактными, контролируемыми и изолированными. Компактность и изолированность изменений упрощает их просмотр, слияние и перебазирование, а также позволяет сохранять историю изменений.
2. *Добавьте свои изменения* с помощью команды git add для каждого изменённого файла.

3. *Зафиксируйте изменения* с помощью команды git commit. Данные фиксации должны соответствовать стандартным соглашениям, приведённым ниже.

   Обязательно включите строку "Signed-off-by:", как это требуется для ядра Linux. Добавление строки означает, что податель согласен с Developer's Certificate of Origin 1.1, текст которого приведён ниже.

   Сертификат происхождения для разработчика, версия 1.1

   Внося свой вклад в проект, я подтверждаю указанное ниже.

   (a) Вклад полностью или частично создан мной и я имею право представить его в соответствии с лицензией для открытого кода, указанной в файле.

   (b) Или вклад основан на предыдущей работе, которая, насколько мне известно, покрывается подходящей лицензией для открытого кода и я имею право в соответствии с этой лицензией представить эту работу с изменениями, независимо от того, созданы ли они полностью мной, на условиях той же лицензии (если мне не разрешено применять иную лицензию), которая указана в файле.

   (c) Вклад был предоставлен мне напрямую другим лицом, которое подтвердило пп. (a), (b) или (c), и я не менял его.

   (d) Я понимаю и согласен с тем, что этот проект и вклад являются общедоступными и запись вклада (включая все предоставленные мной персональные данные) хранится в течение неопределённого времени и может распространяться в соответствии с этим проектом и лицензиями для открытого кода.

   Представьте краткое описание изменений в 1 строку (если требуется более полное описание, его следует включить в тело сообщения). Краткое описание обычно выводится в коротком списке изменений. Если изменения относятся к заданию, следует указать имя задания в качестве префикса краткого описания. В остальных случаях следует указывать в качестве префикса сокращённый путь к изменённому файлу.

   В теле сообщения следует указать более подробные сведения, описывающие суть изменений, их причины и использованный подход. Полезно также привести данные о тестировании правок.

   Если изменение связано с конкретной ошибкой, для которой уже имеется идентификатор отслеживания (bug-tracking ID), следует включить этот идентификатор в подробное описание. Например, в YP применяется соглашение для указания ошибок и фиксациям с устранением конкретной ошибки следует использовать в начале подробного описания строку Fixes [YOCTO #bug-id].

4. *Поместите свои фиксации в репозиторий Contrib,* если у вас есть право записи в него.

   ```
   $ git push upstream_remote_repo local_branch_name
   ```

   Предположим, что у вас есть право добавления в восходящий репозиторий meta-intel-contrib и вы работаете с локальной ветвью your_name/README. Команда git push meta-intel-contrib your_name/README поместит выши локальные фиксации в в репозиторий meta-intel-contrib (ветвь your_name/README).

5. *Определить, кого следует уведомить.* Это может быть сопровождающий или список рассылки, определённый одним из указанных ниже способов.

   (a) *Файл сопровождения* maintainers.inc в каталоге meta/conf/distro/include дерева исходных кодов.

   (b) *Поиск по файлам* с помощью команды git shortlog -- filename, выводящей список фиксаций для указанного файла. Список не упорядочен, но включает всех, кто представлял фиксации.

   (c) *Просмотр списков рассылки* YP и др., перечисленных в разделе Mailing lists [3].

6. *Сделайте запрос на извлечение.* Сообщите сопровождающему или в список рассылки сообщение об отправке изменений, передав запрос на извлечение.

   YP включает два сценария для создания и отправки запросов - create-pull-request и send-pull-request. Эти сценарии хранятся в каталоге scripts дерева исходных кодов (например, ~/poky/scripts). Сценарии корректно формируют запросы без добавления пробелов и тегов HTML. Сопровождающему, который получит запрос напрямую или через список рассылки, сможет сохранить и применить изменения непосредственно из почтового сообщения. Такой метод отправки является предпочтительным.

   Сначала создаётся запрос на извлечение. Например, команда ~/poky/scripts/create-pull-request -u meta-intel-contrib -s "Updated Manual Section Reference in README" запускает сценарий, указывает каталог восходящего репозитория для записи изменений (contrib) и задаёт строку темы в создаваемых patch-файлах. Запуск сценария создаёт файлы правок *.patch в каталоге pull-PID внутри текущего каталога. Одним из файлов является сопроводительное письмо.

   Перед использованием сценария send-pull-request нужно отредактировать сопроводительное письмо, указав сведения об изменениях. Затем можно отправить запрос на извлечение. Например, команда ~/poky/scripts/send-pull-request -p ~/meta-intel/pull-10565 -t meta-intel@yoctoproject.org запускает сценарий и указывает каталог исправления и адрес электронной почты для отправки. Сценарий является интерактивным и нужно следовать выводимым на экран инструкциям.

   Справка об использовании сценариев выводится при указании опции -h.

   ```
   $ poky/scripts/create-pull-request -h
   $ poky/scripts/send-pull-request -h
   ```

3.31.2.2. Использование электронной почты для представления изменений

Можно представлять изменения с помощью электронной почты, отправляя их в соответствующий список рассылки (см. раздел Mailing Lists [3]), как описано ниже.

1. *Внесите изменения локально* в свой репозиторий Git. Изменения следует делать компактными, контролируемыми и изолированными. Компактность и изолированность изменений упрощает их просмотр, слияние и перебазирование, а также позволяет сохранять историю изменений.

2. *Добавьте свои изменения* с помощью команды git add для каждого изменённого файла.

3. *Зафиксируйте изменения с помощью команды* git commit --signoff. Опция --signoff указывает, что изменения внесены вами и подтверждает DCO[1], как описано выше. Данные фиксации должны соответствовать стандартным соглашениям, описанным в п. 3 предыдущего параграфа.

4. *Формат представления* - сообщение электронной почты, создаваемое с помощью команды git format-patch. При вызове команды нужно указать список выпусков или число правок (patch). Например, любая из команда git format-patch -1 и git format-patch HEAD~ берет последнюю одиночную фиксацию (commit) и форматирует её как почтовое сообщение в текущем каталоге. После выполнения команды в текущем каталоге будут созданы нумерованные файлы .patch для представления. При наличии нескольких patch-файлов следует указывать в команде опцию --cover, которая создаёт сопроводительное письмо, как первый patch-файл серии. Это письмо затем можно отредактировать для описания серии правок. Информация о команде git format-patch выводится по команде man git-format-patch.

   Если предполагается частое представление правок для IYP или OE, можно запросить область contrib и связанные с этим права.

5. *Импорт файлов в почтовый клиент* с помощью команды git send-email. Для этого на хосте должна быть установлен и должным образом настроен соответствующий пакет Git (в Ubuntu, Debian и Fedora это git-email).

   Команда git send-email передаёт сообщение с использованием локального или удалённого агента MTA[2], такого как msmtp, sendmail, или через прямую настройку в конфигурационном файле Git ~/.gitconfig. Если правки представляются только по электронной почте, важно отправлять их без пробелов и тегов HTML. Сопровождающему, который получит сообщение нужно его сохранить и применить прямо из почты. Хорошим способом проверки корректности сообщения служит отправка его по своему адресу для тестового сохранения и применения.

   Команда git send-email является предпочтительным методом отправки patch-файлов по электронной почте, поскольку нет риска появления в сообщении ненужных пробелов, которые могут вносить почтовые клиенты. Команда имеет опции для указания получателей и дополнительного редактирования сообщения. Сведения об опциях можно получить с помощью команды man git-send-email.

3.32. Работа с лицензиями

Как отмечено в разделе Licensing [1], проекты с открытым кодом доступны для всех и могут использовать разные лицензии. Здесь описаны механизмы, используемые системой сборки OE для отслеживания изменений в текстах лицензий, и поддержка лицензирования открытого кода в течение жизненного цикла проекта. Рассмотрено также использование в заданиях коммерческих лицензий, которое по умолчанию отключено.

3.32.1. Отслеживание изменений в лицензиях

Лицензия восходящего проекта может с течением времени измениться. Для того, чтобы такие изменения не остались незаметными, переменная LIC_FILES_CHKSUM отслеживает изменения в тексте лицензии. Контрольная сумма проверяется в конце настройки конфигурации и сборка прерывается при несоответствии контрольной суммы.

3.32.1.1. Указание переменной LIC_FILES_CHKSUM

Переменная LIC_FILES_CHKSUM содержит контрольные суммы текста лицензий в исходном коде задания.

```
    LIC_FILES_CHKSUM = "file://COPYING;md5=xxxx \
file://licfile1.txt;beginline=5;endline=29;md5=yyyy \
file://licfile2.txt;endline=50;md5=zzzz \
..."
```

- При использовании beginline и endline следует учитывать, что нумерация строк начинается с 1, а не с 0. Начальная и конечная строка учитываются в контрольной сумме, т. е. в примере будут учитываться строки 1 - 29 в файле licfile1.txt и 1 - 50 в файле licfile2.txt.

- При несоответствии контрольной суммы указанная часть текста лицензии включается в сообщение QA, что позволяет проверить начало и конец учитываемого текста.

Система сборки использует переменную S в качестве принятого по умолчанию каталога при поиске файлов из LIC_FILES_CHKSUM. В примере используются файлы текущего каталога. Рассмотрим ещё один пример.

```
    LIC_FILES_CHKSUM = "file://src/ls.c;beginline=5;endline=16;\
    md5=bb14ed3c4cda583abc85401304b5cd4e"
    LIC_FILES_CHKSUM = "file://${WORKDIR}/license.html;md5=5c94767cedb5d6987c902ac850ded2c6"
```

Первая строка указывает файл ${S}/src/ls.c с учётом строк 5 - 16, вторая указывает файл в WORKDIR. Переменная LIC_FILES_CHKSUM обязательная для всех заданий, где не установлено LICENSE = "CLOSED".

3.32.1.2. Разъяснение синтаксиса

Как отмечено выше, в переменной LIC_FILES_CHKSUM указаны все важные файлы, содержащие текст лицензий для исходного кода. Можно задать контрольную сумму файла целиком или его части, указанной первой и последней учитываемой строкой (полезно для документов, включающих заголовок, файлов README и т. п.). Если параметр beginline или endline опущен, предполагается учёт с первой или до последней строки, соответственно.

[1]Developer's Certificate of Origin.
[2]Mail Transport Agent - агент доставки почты.

Параметр md5 указывает сохранённое значение контрольной суммы md5. Если контрольная сумма отличается от этого значения, лицензия считается изменённой. Если различие обнаруживается при сборке, полученная контрольная сумма md5 помещается в журнал сборки и её можно потом скопировать в задание.

Число файлов, включаемых в LIC_FILES_CHKSUM не ограничивается, однако на практике обычно нужно указывать лишь несколько файлов. Во многих проектах имеется файл COPYING, в котором хранятся сведения о лицензиях для всех файлов исходного кода. Это позволяет отслеживать лишь файл COPYING.

- Если задан пустой или недействительный параметр md5, BitBake возвращает ошибку md5 mis-match и выводит корректное значение параметра в процессе сборки, записывая его также в журнал сборки.
- Если файл содержит лишь текст лицензии, параметры beginline и endline не нужны.

3.32.2. Включение заданий с коммерческими лицензиями
По умолчанию система сборки OE исключает компоненты с коммерческими и иными специальными лицензиями. Это требование определяется на уровне заданий в переменной LICENSE_FLAGS. Например, задание poky/meta/recipes-multimedia/gstreamer/gst-plugins-ugly содержит LICENSE_FLAGS = "commercial". Имеется более сложный пример, содержащий явное имя и версию задания (после преобразования переменной).

```
LICENSE_FLAGS = "license_${PN}_${PV}"
```

Для включения в образ компоненты, ограниченной определением LICENSE_FLAGS, нужно добавить соответствующую запись в переменную LICENSE_FLAGS_WHITELIST, которая обычно задаётся в файле local.conf. Например, для включения пакета poky/meta/recipes-multimedia/gstreamer/gst-plugins-ugly можно добавить строку commercial_gst-plugins-ugly или более общий вариант commercial в переменную LICENSE_FLAGS_WHITELIST. Полное описание работы переменной приведено в параграфе 3.32.2.1. Соответствие флагов лицензий.

Для дополнительного включения пакета, собранного из задания с LICENSE_FLAGS = "license_${PN}_${PV}", в предположении, что файл задания называется emgd_1.10.bb в следующем примере добавлено license_emgd_1.10.

```
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly license_emgd_1.10"
```

Не требуется указывать полную строку лицензии в списке разрешений и можно применять сокращённую форму, которая состоит из первой части строки лицензии до первого символа _. Сокращённая строка будет соответствовать любой лицензии, содержащей эту подстроку. Например, LICENSE_FLAGS_WHITELIST = "commercial license" будет соответствовать обоим пакетам, упомянутым выше, а также все прочим пакетам с лицензией, начинающейся с commercial или license.

3.32.2.1. Соответствие флагов лицензий
Соответствие флагов лицензий позволяет контролировать задания, включаемый в сборку системой OE. По сути система сборки пытается сопоставить строки LICENSE_FLAGS из заданий со строками LICENSE_FLAGS_WHITELIST и при совпадении включает задание в сборку, а при различии исключает. Сопоставление флагов в общем случае просто, однако понимание некоторых нюансов облегчит его использование.

Перед сравнением флага, определённого конкретным заданием, с содержимым разрешённого списка к этому флагу добавляется преобразованная переменная _${PN}. Преобразование делает каждое значение LICENSE_FLAGS зависящим от задания. Например, указание LICENSE_FLAGS = "commercial" в задании foo ведёт к строке commercial_foo. Соответствующая строка сравнивается со списком разрешений.

Разумное применение строк LICENSE_FLAGS и содержимого LICENSE_FLAGS_WHITELIST обеспечивает гибкость включения и исключения заданий на основе лицензии. Например, можно расширить сопоставление использованием подстрок флагов лицензий. В этом случае нужно применять ту часть преобразованной строки, которая предшествует добавленному символу _ (например usethispart для usethispart_1.3, usethispart_1.4 и т. п.).

Например, простое указание строки commercial в списке разрешений, будет соответствовать любому преобразованному определению LICENSE_FLAGS, начинающемуся с commercial (commercial_foo, commercial_bar), которое автоматически создаётся системой сборки для гипотетических заданий foo и bar в предположении, что эти задания включают LICENSE_FLAGS = "commercial".

Таким образом можно задать исчерпывающий список флагов лицензий в списке разрешений и разрешить использование в образе только определённых заданий или использовать подстроки для расширения совпадений, чтобы разрешить включение более широкого спектра заданий. Эта схема работает даже в тех случаях, когда к строке LICENSE_FLAGS уже добавлен суффикс _${PN}. Например, система сборки преобразует флаг лицензии commercial_1.2_foo в commercial_1.2_foo_foo и задание будет соответствовать в списке разрешений строкам commercial и commercial_1.2_foo.

Ниже приведены несколько других вариантов.

- Можно указать в задании foo строку с версией, такую как commercial_foo_1.2. Система сборки преобразует эту строку в commercial_foo_1.2_foo. Комбинация этого флага со списком разрешений, включающим commercial, обеспечит совпадение как и для любого другого флага, начинающегося со строки commercial.
- В некоторый случаях можно использовать commercial_foo в списке разрешений совпадать будет не только commercial_foo_1.2, но и любой флаг, начинающийся с commercial_foo, независимо от версии.
- Можно задать в списке разрешений пакет и версию (например, commercial_foo_1.2) для точного совпадения.

3.32.2.2. Другие варианты, связанные с коммерческими лицензиями
Другие полезные переменные для работы с коммерческими лицензиями определены в файле poky/meta/conf/distro/include/default-distrovars.inc.

```
COMMERCIAL_AUDIO_PLUGINS ?= ""
COMMERCIAL_VIDEO_PLUGINS ?= ""
```

Для включения этих компонент можно указать в файле local.conf строки вида

```
COMMERCIAL_AUDIO_PLUGINS = "gst-plugins-ugly-mad \
   gst-plugins-ugly-mpegaudioparse"
COMMERCIAL_VIDEO_PLUGINS = "gst-plugins-ugly-mpeg2dec \
   gst-plugins-ugly-mpegstream gst-plugins-bad-mpegvideoparse"
LICENSE_FLAGS_WHITELIST = "commercial_gst-plugins-ugly commercial_gst-plugins-bad commercial_qmmp"
```

Можно конечно создать для этих компонент список разрешений LICENSE_FLAGS_WHITELIST = "commercial", но он включит и другие пакеты, где LICENSE_FLAGS содержит commercial, что может оказаться нежелательным. Указание подключаемых модулей в операторах COMMERCIAL_AUDIO_PLUGINS и COMMERCIAL_VIDEO_PLUGINS (вместе с разрешением LICENSE_FLAGS_WHITELIST) включает плагины или компоненты в сборку образа.

3.32.3. Поддержка соответствия лицензиям в жизненном цикле проекта

Одной из задач разработки программ с открытым кодом является обеспечение соответствия требованиям лицензий для открытого кода в течение жизненного цикла программы. В этом разделе нет юридических консультаций и не рассматриваются все варианты, но описаны некоторые методы, которые позволят обеспечить соответствие требованиям при выпуске программ.

Среди сотен разных лицензий, отслеживаемых YP, сложно знать требования каждой из них. Однако требования основных лицензий FLOSS могут выполняться, если соблюдаются три главных условия:

- предоставлен исходный код программы;
- предоставлен текст лицензии для программы;
- предоставлены сценарии компиляции и изменения исходного кода.

Имеются и другие требования, а также методы, описанные здесь (например, механизм распространения исходного кода). Поскольку в разных организациях применяются различные методы соблюдения лицензий, здесь не описывается универсальный способ выполнить требования, а рассматриваются методы обеспечения соответствия путём выполнения трёх приведённых выше условий. После их соблюдения перед выпуском образа, исходных кодов или системы сборки следует проверить полноту всех компонент. В процессе сборки образа YP создаёт манифест лицензий, размещаемый в каталоге ${DEPLOY_DIR}/licenses/image_name-datestamp, который поможет при проверке.

3.32.3.1. Предоставление исходного кода

Работа по обеспечению соответствия должна начинаться до создания финального образа. Первым делом нужно обратить внимание на предоставление исходного кода. YP обеспечивает несколько способов решения этой задачи.

Одним из простейших способов является предоставление всего каталога DL_DIR, используемого для сборки. Однако этот метод имеет ряд недостатков и прежде всего размер, поскольку каталог включает все исходные коды, используемые при сборке, а не только код выпускаемого образа. В каталоге размещены исходные коды инструментария, который обычно не включается в образ. Но более серьёзной проблемой может стать непредусмотренный выпуск кода фирменных программ. Для устранения проблемы YP поддерживает класс archiver.

Прежде чем использовать DL_DIR или класс archiver, нужно решить вопрос о способе представления кода. Класс archiver может создавать архивы и SRPM с разными уровнями соответствия. Один из способов заключается просто в выпуске архива (tarball) исходного кода. Это можно сделать, добавив в файл local.conf каталога сборки строки вида

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
```

В процессе создания образа исходный код всех включённых в образ заданий помещается в каталоги DEPLOY_DIR/sources на основе переменной LICENSE для каждого задания. Выпуск каталога целиком позволяет выполнить требования по предоставлению исходного кода без изменений. Важно отметить, что размер каталога может оказаться большим. Решением проблемы размера может быть выпуск лишь архивов, для которых лицензия требует предоставление кода. Предположим, что нужен лишь код GPL, определяемый приведённым ниже сценарием.

```
# Сценарий для архивирования пакетов с определёнными требованиями лицензий
# Файлы исходного кода и лицензий копируются в каталоги пакетов
# Сценарий нужно запускать из каталога сборки (build)
#!/bin/bash
src_release_dir="source-release"
mkdir -p $src_release_dir
for a in tmp/deploy/sources/*; do
   for d in $a/*; do
      # Получить имя пакета из пути
      p=`basename $d`
      p=${p%-*}
      p=${p%-*}
      # Архивировать только пакеты GPL (измените *GPL* для других лицензий)
      numfiles=`ls tmp/deploy/licenses/$p/*GPL* 2> /dev/null | wc -l`
      if [ $numfiles -gt 1 ]; then
          echo Archiving $p
          mkdir -p $src_release_dir/$p/source
          cp $d/* $src_release_dir/$p/source 2> /dev/null
          mkdir -p $src_release_dir/$p/license
          cp tmp/deploy/licenses/$p/* $src_release_dir/$p/license 2> /dev/null
      fi
   done
done
```

На этом этапе можно создать архив из каталога gpl_source_release и предоставить его пользователям. Этот метод является этапом соблюдения разделов 3a в GPLv2 и 6 в GPLv3.

3.32.3.2. Предоставление текста лицензий

Одним из требований, которое часто игнорируют, является включение текста лицензии. Это требование тоже нужно выполнить до создания финального образа. Некоторые лицензии требуют предоставлять их текст вместе с двоичными файлами. Это можно сделать, добавив в файл local.conf строки вида

```
COPY_LIC_MANIFEST = "1"
COPY_LIC_DIRS = "1"
LICENSE_CREATE_PACKAGE = "1"
```

В результате тексты лицензий в процессе сборки будут включены в образ. Установка значения 1 для всех трёх переменных приводит к наличию двух копий файла лицензии (в /usr/share/common-licenses и /usr/share/license). Это обусловлено тем, что переменные COPY_LIC_DIRS и COPY_LIC_MANIFEST добавляют копию лицензии при сборке образа, но не предлагают путь добавления лицензий для недавно установленных в образ пакетов. Переменная LICENSE_CREATE_PACKAGE добавляет отдельный пакет и путь обновления при добавлении лицензий в образ.

Поскольку класс archiver уже заархивировал исходный код без изменений, содержащий файлы лицензий, требования по представлению лицензии с исходным кодом, заданные GPL и другими лицензиями для открытого кода, выполнены.

3.32.3.3. Предоставление сценариев компиляции и изменений исходного кода

После выполнения рекомендаций предыдущих параграфов можно собирать образ. Выпуск версии системы сборки OE и использованных при сборке уровней выполняет сразу требования к сценариям компиляции и изменениям кода.

При наличии уровней BSP и дистрибутива эти уровни используются для правки, компиляции, упаковки или изменения (если оно имеется) всех программ с открытым кодом, включённых в образ, может потребоваться выпуск этих уровней в соответствии с разделом 3 лицензии GPLv2 или разделом 1 лицензии GPLv3. Одним из способов решения задачи является явный выбор версии YP и уровней, используемых при сборке. Например,

```
# Сборка с использованием ветви warrior репозитория poky
$ git clone -b warrior git://git.yoctoproject.org/poky
$ cd poky
# Сборка с использованием release_branch для уровней
$ git clone -b release_branch git://git.mycompany.com/meta-my-bsp-layer
$ git clone -b release_branch git://git.mycompany.com/meta-my-software-layer
# Очистка репозиториев .git
$ find . -name ".git" -type d -exec rm -rf {} \;
```

Для удобства конечных пользователей разработчики могут рассмотреть изменение файла meta-poky/conf/bblayers.conf.sample, чтобы обеспечить автоматическое включение созданных организацией уровнем уровней при использовании выпущенной системы сборки для создания образа, как показано ниже.

```
# POKY_BBLAYERS_CONF_VERSION каждый раз увеличивается в build/conf/bblayers.conf
# изменения несовместимы
POKY_BBLAYERS_CONF_VERSION = "2"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  ##OEROOT##/meta \
  ##OEROOT##/meta-poky \
  ##OEROOT##/meta-yocto-bsp \
  ##OEROOT##/meta-mylayer \
  "
```

Создание и предоставление архива метаданных уровней (задания, файлы конфигурации и т. п.) обеспечивает соблюдение требований по включению сценариев управления компиляцией и всех изменений оригинального кода.

3.32.4. Копирование отсутствующих лицензий

Некоторые пакеты (например, linux-firmware) могут иметь лицензии, которые мало распространены. Можно избежать использования множества лицензий общего назначения, применимых лишь к определенным пакетам, путём использования переменной NO_GENERIC_LICENSE. Это также позволяет избежать ошибок QA при использовании в задании необычной, но и не закрытой лицензии. Ниже приведён пример использования файла LICENSE.Abilis.txt в качестве лицензии для извлечённого исходного кода.

```
NO_GENERIC_LICENSE[Firmware-Abilis] = "LICENSE.Abilis.txt"
```

3.33. Использование инструмента отчётов об ошибках

Инструмент отчётов об ошибках позволяет представлять ошибки, возникшие в процессе сборки, в центральную базу данных. Извне системы сборки можно использовать web-интерфейс базы для поиска и просмотра ошибок, а также статистики. Инструмент использует модель «клиент-сервер» с реализацией клиентской части в дереве исходных кодов YP (например, poky). Сервер получает данные от клиента и записывает их в базу.

Действующий сервер отчётов об ошибках доступен по ссылке http://errors.yoctoproject.org. Этот сервер организован для того, чтобы можно было получить помощь при возникновении ошибок и предоставить полную информацию об ошибке, а затем указать ссылку на неё (URL) и отправить сообщение в список рассылки. Отправленные на сервер отчёты доступны для всех.

3.33.1. Включение отчётов

По умолчанию инструмент отчётов об ошибках отключён и для его включения нужно наследовать класс report-error, добавив в конце файла local.conf в каталоге сборки строку INHERIT += "report-error". Данные отчётов по умолчанию записываются в файл ${LOG_DIR}/error-report, однако можно задать другое место хранения командой вида ERR_REPORT_DIR = "path".

Включение отчётов об ошибках заставляет систему сборки фиксировать все сообщения и записывать их в указанный файл. При возникновении ошибки система сборки включает в консольный вывод команду для отправки файла с информацией на сервер. Например, для передачи информации на восходящий сервер может служить команда $ send-error-report /home/brandusa/project/poky/build/tmp/log/error-report/error_report_201403141617.txt, которая отправит информацию в общедоступную базу данных на сервере http://errors.yoctoproject.org. При указании конкретного сервера сведения можно отправить в другую базу данных. Информация о работе с инструментом доступна по команде send-error-report --help.

При отправке файла выводится приглашение на просмотр передаваемых данных, а также указание имени и (необязательного) почтового адреса. После ввода информации команда возвращает идентификатор переданных сведений на сервере, например, http://errors.yoctoproject.org/Errors/Details/9522/, который можно использовать для последующей работы с этой ошибкой.

3.33.2. Отключение отчётов

Для отключения отчётов об ошибках следует указать в конце файла local.conf строку INHERIT += "report-error".

3.33.3. Установка своего сервера отчётов об ошибках

При желании можно установить свой сервер отчётов об ошибках, загрузив его код из репозитория Git http://git.yoctoproject.org/cgit/cgit.cgi/error-report-web/. Установка сервера описана в файле README.

3.34. Использование Wayland и Weston

Wayland - протокол сервера отображения, обеспечивающий менеджерам окон возможность прямого взаимодействия с приложениями и видеосистемой при использовании других библиотек. Использование Wayland с поддерживающими протокол платформами может обеспечить лучшее управления разбором графических кадров.

YP включает библиотеки протокола Wayland и эталонный сборщик Weston, размещая их в дереве источников. В частности, задания для Wayland и Weston находятся в каталоге meta/recipes-graphics/wayland. Можно собрать пакеты Wayland и Weston для использования с платформами, принимающими инфраструктуру Mesa 3D и Direct Rendering (Mesa DRI). Это означает, что вы не сможете собрать и использовать пакеты, если ваша платформа поддерживает, например, Intel® Embedded Media и Graphics Driver (Intel® EMGD), которые переопределяют Mesa DRI.

По причине отсутствия поддержки EGL пакет Weston 1.0.3 не работает напрямую на эмулируемом в QEMU оборудовании. Однако эта версия Weston без проблем работает с X=эмуляцией.

3.34.1. Включение Wayland в образе

Для включения Wayland нужно добавить его в сборку и установить в образ.

3.34.1.1. Сборка

Чтобы заставить Mesa собрать платформу wayland-egl, а Weston - собрать Wayland с поддержкой KMS[1], нужно добавить флаг wayland в оператор DISTRO_FEATURES в файле local.conf в форме DISTRO_FEATURES_append = " wayland". Если где-нибудь включена поддержка X11, Weston будет собирать Wayland с поддержкой X11.

3.34.1.2. Установка

Для установки Wayland в образ нужно включить в файл local.conf оператор CORE_IMAGE_EXTRA_INSTALL += "wayland weston".

3.34.2. Запуск Weston

Для запуска Weston в среде X11 нужно включить пакет, как описано выше и собрать образ Sato. В этом случае Weston Launcher помещается в категорию Utility.

Можно запустить Weston из командной строки, что зачастую удобней в среде разработки. Для этого следует выполнить указанные ниже действия.

1. Ввести для экспорта XDG_RUNTIME_DIR команды

```
mkdir -p /tmp/$USER-weston
chmod 0700 /tmp/$USER-weston
export XDG_RUNTIME_DIR=/tmp/$USER-weston
```

2. Запустить Weston из оболочки командой weston.

Глава 4. Использование QEMU

YP использует открытую реализацию эмулятора QEMU[2] как часть инструментария. В этой главе рассмотрена работа с эмулятором и его применение в разработке.

4.1. Обзор

В контексте YP эмулятор и машина виртуализации QEMU позволяет запускать полные образы, собранные в YP, как задачи системы сборки. Эмулятор QEMU полезен для запуска и тестирования образов и приложений для поддерживаемой YP архитектуры без реального оборудования. YP также использует QEMU для запуска автоматизированных тестов QA на финальных образах, включаемых в каждый выпуск. Реализация эмулятора несколько отличается от обычного QEMU и здесь приведён её краткий обзор. Официальная информация доступна на сайте *QEMU* и в *руководстве пользователя*.

[1]Kernel Mode Setting - установка режима ядра.
[2]Quick EMUlator.

4.2. Запуск QEMU

Для использования QEMU нужно установить и инициализировать эмулятор, а также иметь нужные для работы элементы (образы и файловые системы). Ниже описаны этапы подготовки к работе с QEMU.

1. *Установка QEMU.* Эмулятор можно сделать доступным в YP разными способами, одним из которых является установка SDK (см. раздел The QEMU Emulator [2]).
2. *Настройка среды.*
   - При выборе репозитория poky с загрузкой и распаковкой архива YP можно организовать среду сборки приведёнными ниже командами.

     ```
     $ cd ~/poky
     $ source oe-init-build-env
     ```
   - При установке кросс-инструментов можно запустить сценарий их инициализации. Например, приведённая ниже команда запускает сценарий из принятого по умолчанию каталога poky_sdk.

     ```
     . ~/poky_sdk/environment-setup-core2-64-poky-linux
     ```
3. *Проверка наличия нужных элементов.* Нужно убедиться в наличии собранного ядра, которое будет загружаться в QEMU, а также корневой файловой системы для целевой машины и архитектуры.
   - Если был собран образ для QEMU (qemux86, qemuarm и т. п.), его элементы будут находиться в каталоге сборки.
   - Если образа ещё нет, нужно зайти на страницу machines/qemu и загрузить готовый образ для целевой машины и архитектуры.

   Извлечение корневой файловой системы описано в разделе Extracting the Root Filesystem [2].
4. *Запуск QEMU* командой вида $ runqemu [option] […]. В соответствии с введённой командой runqemu выполнит заданные действия. Например, по умолчанию QEMU ищет собранный последним образ по временной метке. В параметрах нужно указать по меньшей мере машину, образ виртуальной машины (*wic.vmdk) или образ ядра (*.bin). Ниже приведено несколько примеров запуска QEMU:
   - Запуск QEMU с MACHINE = "qemux86". В предположении стандартного каталога сборки runqemu автоматически найдёт образ bzImage-qemux86.bin и файловую систему core-image-minimal-qemux86-20140707074611.rootfs.ext3 (для образа core-image-minimal). При наличии нескольких образов с одним именем, QEMU использует более свежий.

     ```
     $ runqemu qemux86
     ```
   - Похож на предыдущий пример, но явно указывается образ и тип корневой файловой системы.

     ```
     $ runqemu qemux86 core-image-minimal ext3
     ```
   - Загрузка образа initramfs для включения звука в QEMU. В этом случае runqemu устанавливает внутреннюю переменную FSTYPE = "cpio.gz". Для включения звука должен быть установлен подходящий драйвер.

     ```
     $ runqemu qemux86 ramfs audio
     ```
   - В этом примере не представлено информации, достаточной для запуска QEMU. Хотя корневая файловая система задана, нужно ещё указать хотя бы MACHINE, KERNEL или VM.

     ```
     $ runqemu ext3
     ```
   - Этот пример задаёт загрузку образа виртуальной машины (файл .wic.vmdk). Из .wic.vmdk программа runqemu определяет архитектуру QEMU (MACHINE) qemux86 и корневую файловую систему vmdk.

     ```
     $ runqemu /home/scott-lenovo/vm/core-image-minimal-qemux86.wic.vmdk
     ```

4.3. Переключение консоли

При загрузке и работе QEMU можно переключаться между поддерживаемыми консолями с помощью клавиш Ctrl+Alt+цифра. Например, Ctrl+Alt+3 переключает на последовательную консоль, если она активна. Возможность переключения полезна, например, при нарушении по какой-либо причине работы основной консоли QEMU. Обычно 2 служит для переключения на основную консоль, 3 - на последовательную.

4.4. Удаление заставки

Можно удалить заставку при загрузке QEMU с помощью клавиш Alt+<-. Это позволяет увидеть фоновый вывод.

4.5. Запрет захвата курсора

Используемая по умолчанию интеграция QEMU захватывает курсор в главном окне. Это обусловлено тем, что стандартные мыши обеспечивают только относительные перемещения, а не абсолютные координаты. Можно отменит захват курсора с помощью клавиш Ctrl+Alt. Интеграция QEMU в YP поддерживает сенсорные панели wacom USB, которые обеспечивают абсолютные координаты, что позволяет курсору входит в главное окно и выходить из него без захвата, упрощая работу пользователя.

4.6. Запуск на сервере NFS

Одним из вариантов работы QEMU является запуск на сервере NFS. Это полезно в тех случаях, когда нужно обращаться к одной файловой системы с хоста сборки и эмулируемой системы. Следует отметить, что для запуска здесь не нужны полномочия root, поскольку применяется сервер NFS в пользовательском пространстве. Ниже описана процедура запуска QEMU с использованием сервера NFS.

1. *Извлечение корневой файловой системы.* Когда все готово к запуску QEMU в среде, можно использовать сценарий runqemu-extract-sdk из каталога scripts. Сценарий runqemu-extract-sdk принимает архив корневой файловой системы и распаковывает его в указанное место. Например, команда runqemu-extract-sdk

./tmp/deploy/images/qemux86/core-image-sato-qemux86.tar.bz2 test-nfs распаковывает файловую систему в каталог test-nfs.

2. *Запуск QEMU.* После извлечения корневой файловой системы можно запустить runqemu, указав расположение файловой системы. При этом изменения, внесённые в каталог ./test-nfs будут видны при работе. Например, для работы с образом qemux86 служит команда runqemu qemux86 ./test-nfs.

Для запуска, остановки или перезапуска общего ресурса NFS служат приведённые ниже команды.

- Запуск общего раздела NFS - runqemu-export-rootfs start file-system-location.
- Остановка общего раздела NFS - runqemu-export-rootfs stop file-system-location.
- Перезапуск общего раздела NFS - runqemu-export-rootfs restart file-system-location.

4.7. Совместимость QEMU с CPU при работе с KVM

По умолчанию сборка QEMU компилируется для 64-битовых и x86 Intel® Core™2 Duo процессоров, а также 32-битовых x86 Intel® Pentium® II. QEMU собирается для этих CPU по причини их совместимости с широким спектром менее распространённых CPU.

Однако, несмотря на широкую совместимость, эти процессоры могут использовать функции, не поддерживаемые процессором вашего хоста. Это не вызывает проблем при использовании в QEMU программной эмуляции функций, но проблемы могут возникать при работе QEMU со включённым KVM. В частности, программы, скомпилированные для некоторых CPU, не могут работать на CPU под управлением KVM без поддержки функции. Для решения проблемы можно переопределить настройку CPU в QEMU путём установки переменной QB_CPU_KVM в файле qemuboot.conf в каталоге deploy/image. Эта установка задаёт опцию -cpu, передаваемую QEMU в сценарии runqemu. Для просмотра списка поддерживаемых типов служит команда qemu -cpu.

4.8. Производительность QEMU

При использовании QEMU для эмуляции оборудования могут возникать проблемы производительности в зависимости от комбинации целевой системы и архитектуры. Например, работа образа qemux86 на 32-битовом хосте Intel x86 достаточно производительна в результате соответствия архитектуры целевой системы и хоста. При использовании образа qemuarm на том же хосте Intel производительность может упасть. Но специфические аспекты ARM будут эмулироваться достаточно точно.

Для ускорения работы образы QEMU поддерживают использование distcc для вызова кросс-компилятора вне эмулируемой системы. Если для запуска QEMU применяется runqemu и на хосте имеется приложение distccd, любой инструмент кросс-компиляции BitBake, доступный системе сборки, может автоматически использоваться из QEMU просто путём вызова distcc. Это можно сделать, определив переменную кросс-компилятора (например, export CC="distcc"). При использовании подходящего образа SDK или автономного инструментария можно также применять эти инструменты.

Имеется несколько механизмов для подключения к системе, работающей на эмуляторе QEMU.

- QEMU предоставляет интерфейс framebuffer, делающий доступными стандартные консоли.
- Обычно устройства без монитора имеют стандартный последовательный порт. В этом случае можно настроить операционную систему работающего образа для использования последовательной консоли. Соединение использует стандартную сеть IP.
- Для некоторых образов QEMU имеются серверы SSH. В образе core-image-sato QEMU сервер Dropbear (SSH) работает без доступа пользователя root. Образы core-image-full-cmdline и core-image-lsb используют OpenSSH вместо Dropbear. Эти серверы поддерживают стандартные команды ssh и scp. Однако в core-image-minimal сервера SSH нет.
- Можно использовать сервер NFS в пользовательском пространстве для загрузки сессии QEMU с использованием локальной копии корневой файловой системы на хосте. Для организации соединения нужно распаковать архив корневой файловой системы с помощью команды runqemu-extract-sdk, а после этого указать сценарию runqemu распакованный каталог вместо образа коревой файловой системы (см. раздел 4.6. Запуск на сервере NFS).

4.9. Синтаксис командной строки QEMU

Базовый синтаксис команды runqemu имеет форму runqemu [option] […]. В соответствии с введённой командой runqemu выполнит заданные действия. Например, по умолчанию QEMU ищет собранный последним образ по временной метке. В параметрах нужно указать по меньшей мере машину, образ виртуальной машины (*wic.vmdk) или образ ядра (*.bin). Ниже приведён вывод команды с опцией --help, описывающий команды runqemu.

```
$ runqemu --help

Usage: you can run this script with any valid combination
of the following environment variables (in any order):
  KERNEL - the kernel image file to use
  ROOTFS - the rootfs image file or nfsroot directory to use
  MACHINE - the machine name (optional, autodetected from KERNEL filename if unspecified)
  Simplified QEMU command-line options can be passed with:
    nographic - disable video console
    serial - enable a serial console on /dev/ttyS0
    slirp - enable user networking, no root privileges is required
    kvm - enable KVM when running x86/x86_64 (VT-capable CPU required)
    kvm-vhost - enable KVM with vhost when running x86/x86_64 (VT-capable CPU required)
    publicvnc - enable a VNC server open to all hosts
    audio - enable audio
```

```
    [*/]ovmf* - OVMF firmware file or base name for booting with UEFI
  tcpserial=<port> - specify tcp serial port number
  biosdir=<dir> - specify custom bios dir
  biosfilename=<filename> - specify bios filename
  qemuparams=<xyz> - specify custom parameters to QEMU
  bootparams=<xyz> - specify custom kernel parameters during boot
  help, -h, --help: print this text

Examples:
  runqemu
  runqemu qemuarm
  runqemu tmp/deploy/images/qemuarm
  runqemu tmp/deploy/images/qemux86/<qemuboot.conf>
  runqemu qemux86-64 core-image-sato ext4
  runqemu qemux86-64 wic-image-minimal wic
  runqemu path/to/bzImage-qemux86.bin path/to/nfsrootdir/ serial
  runqemu qemux86 iso/hddimg/wic.vmdk/wic.qcow2/wic.vdi/ramfs/cpio.gz...
  runqemu qemux86 qemuparams="-m 256"
  runqemu qemux86 bootparams="psplash=false"
  runqemu path/to/<image>-<machine>.wic
  runqemu path/to/<image>-<machine>.wic.vmdk
```

4.10. Опции команды runqemu

Ниже приведено описание опций команды runqemu. При некорректной комбинации опций или их нехватке runqemu выдаёт сообщение, помогающее найти и устранить проблему.

- *QEMUARCH* - архитектура машины QEMU (qemuarm, qemuarm64, qemumips, qemumips64, qemuppc, qemux86, qemux86-64).
- *VM* - образ виртуальной машины (.wic.vmdk) для загрузки. Имя файла должно включать qemux86-64, qemux86, qemuarm, qemumips64, qemumips, qemuppc или qemush4.
- *ROOTFS* - корневая файловая система ext2, ext3, ext4, jffs2, nfs или btrfs. Для nfs нужен полный путь к корню.
- *KERNEL* - образ ядра (.bin). При наличии файла .bin runqemu обнаруживает его и считает образом ядра.
- *MACHINE* - архитектура машины QEMU (qemux86, qemux86-64, qemuarm, qemuarm64, qemumips, qemumips64, qemuppc). Опции MACHINE и QEMUARCH в основном идентичны. Если опция MACHINE не задана, runqemu пытается определить машину из других опций.
- *ramfs* указывает загрузку образа initramfs, означающего тип файловой системы (FSTYPE) cpio.gz.
- *iso* указывает загрузку образа ISO, означающего тип файловой системы (FSTYPE) .iso.
- *nographic* отключает графическую консоль, устанавливая консоль на ttys0. Опция полезна, когда вы подключены к серверу и не хотите отключать пересылку X11 на рабочую станцию.
- *serial* включает последовательную консоль на порту /dev/ttyS0.
- *biosdir* устанавливает пользовательский каталог для BIOS, VGA BIOS и keymap.
- *biosfilename* устанавливает пользовательское имя BIOS.
- *qemuparams=\"xyz\"* задаёт пользовательские параметры QEMU и служит для передачи опций, отличных от kvm и serial.
- *bootparams=\"xyz\"* задаёт пользовательские параметры для загрузки ядра.
- *audio* включает звук в QEMU. Опция MACHINE при этом должна иметь значение qemux86 или qemux86-64. Кроме того, нужен драйвер snd_intel8x0 или snd_ens1370 на гостевой системе linux.
- *slirp* включает сеть slirp, для которой не требуется доступ root, но использование сложнее и менее функционально.
- *kvm* включает KVM для архитектуры qemux86 или qemux86-64. Для работы KVM нужно выполнить 3 условия:
  - опция MACHINE должна иметь значение qemux86 или qemux86-64;
  - на хосте сборки должны быть установлены модули KVM (/dev/kvm);
  - каталог /dev/kvm на сборочном хосте должен быть открыт для чтения и записи.
- *kvm-vhost* включает KVM с поддержкой VHOST для архитектуры qemux86 или qemux86-64. Для работы KVM с VHOST нужно выполнить 3 условия:
  - должны быть выполнены условия для опции kvm;
  - на хосте сборки должно быть устройство virtio net (/dev/vhost-net);
  - каталог /dev/vhost-net на сборочном хосте должен быть открыт для чтения и записи, а также требуется поддержка slirp;
- *publicvnc* включает сервер VNC на всех хостах.

Литература

[1] Yocto Project Overview and Concepts Manual, https://www.yoctoproject.org/docs/2.7.1/overview-manual/overview-manual.html.

[2] Yocto Project Application Development and the Extensible Software Development Kit (eSDK), http://www.yoctoproject.org/docs/2.7.1/sdk-manual/sdk-manual.html.

[3] Yocto Project Reference Manual, http://www.yoctoproject.org/docs/2.7.1/ref-manual/ref-manual.html.

[4] Yocto Project Quick Build, http://www.yoctoproject.org/docs/2.7.1/brief-yoctoprojectqs/brief-yoctoprojectqs.html.

[5] Yocto Project Board Support Package (BSP) Developer's Guide, https://www.yoctoproject.org/docs/2.7.1/bsp-guide/bsp-guide.html.

[6] BitBake User Manual, http://www.yoctoproject.org/docs/2.7.1/bitbake-user-manual/bitbake-user-manual.html.

[7] Yocto Project Linux Kernel Development Manual, https://www.yoctoproject.org/docs/2.7.1/kernel-dev/kernel-dev.html.

[8] Toaster User Manual, http://www.yoctoproject.org/docs/2.7.1/toaster-manual/toaster-manual.html.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru