Yocto Project Overview and Concepts Manual

Scott Rifenbark

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

Этот документ основан на переводе *Yocto Project Overview and Concepts Manual* для выпуска 3.0 Yocto Project. Свежие версии оригинальных документов можно найти на странице документации Yocto Project. Размещённые там материалы более актуальны, нежели включённые в архивы пакета Yocto Project.

Оглавление

Глава 1. Обзор и концепции Yocto Project

1.1. Введение

В этом документе рассматриваются основные концепции Yocto Project (YP). Приведен также обзор программных компонент, широко применяющихся методов и другие базовые сведения для новых пользователей YP.

- *Глава 2. Введение в YP* содержит вводные сведения о YP, описание возможностей и сложностей YP, модели уровней, компонент и инструментов, методов разработки, эталонного дистрибутива Poky, системы сборки OpenEmbedded (OE) и некоторых базовых терминов Yocto.
- *Глава 3. Среда разработки YP* поможет обрести понимание процессов разработки в YP. Здесь приведена информация о программах с открытым кодом, хосте для разработки, репозиториях исходных кодов YP, процессах работы с Git и YP, пример Git и сведения о лицензировании.
- *Глава 4. Концепции YP описывает связанные с* YP концепции и поможет найти информацию о компонентах, разработке, кросс-инструментах и т. п.

Для более глубокого ознакомления следует обратиться к дополнительным источникам, перечисленным ниже.

- *Пошаговые инструкции для задач разработки* содержатся в других документах YP. В руководстве для разработчиков [1] приведены примеры решения различных задач разработки. Руководство для разработчиков SDK [2] содержит подробные сведения об установке SDK, используемых при разработке приложений.
- *Справочные руководства* включают справочник по YP [3] и справочник по разработке BSP[1] [4].

1.2. Дополнительная информация

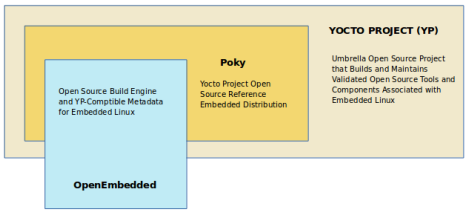
Этот документ содержит базовую информацию по разным темам, поэтому для лучшего понимания будет полезно обратиться к дополнительным источникам. Сведения от YP можно получить на сайте Yocto Project. Краткая информация по сборке образов без углубления в YP приведена в документе [5]. В разделе Links and Related Documentation [3] приведены ссылки на дополнительные документы.

Глава 2. Введение в YP

2.1. Что такое YP

YP представляет собой проект с открытым исходным кодом, помогающий разработчикам создавать специализированные системы на основе Linux для встраиваемых платформ, независимо от аппаратной архитектуры. YP предоставляет широкий набор инструментов и среду разработки, позволяющие сотрудничать разработчикам встраиваемых систем на основе общих технологий, программных стеков, конфигураций и обобщения опыта для создания полнофункциональных образов Linux.

Тысячи разработчиков по всему миру оценили преимущества YP в части разработки системных и прикладных программ, архивирования и поддержки, а также настройки под свои задачи, обеспечивающие высокую скорость, минимальный объем и экономию памяти. Проект является фактическим стандартом для создания программных стеков и позволяет настраивать конфигурацию программ и создавать переносимые между разными аппаратными платформами образы, которые достаточно просты в поддержке и расширении.



Дополнительная вводная информация приведена в статье и коротком видеоролике.

2.1.1. Возможности

- *Широкое распространение в отрасли.* Многие производители микросхем, операционных систем, программ и поставщики услуг приспособили свою продукцию и услуги к использованию YP. Сообщество и компании, вовлечённые в YP, представлены на вкладках COMMUNITY и ECOSYSTEM сайта Yocto Project.
- *Независимость от архитектуры.* YP поддерживает архитектуру Intel, ARM, MIPS, AMD, PPC и др. Большинство ODM, OSV и производителей микросхем создали и поддерживают пакеты BSP для своего оборудования. При использовании своих микросхем можно создать специальные пакеты BSP. Кроме того, YP поддерживает эмуляцию множества устройств на основе QEMU[2].

[1]Board Support Package - пакет поддержки платы.
[2]Quick EMUlator.

- *Простота переноса образов и кода.* Вывод YP легко перенести из одной архитектуры в другую без смены среды разработки. Кроме того, если вы не можете обеспечить поддержку созданных образов или приложений, её можно передать на поддержку коммерческим производителям Linux, таким как Wind River, Mentor Graphics, Timesys, ENEA.

- *Гибкость.* Компании применяют YP множеством разных способов. Например, некоторые создают свои дистрибутивы Linux в качестве базы кода для разной продукции. Возможности настройки и поддержка уровней может упростить создание базового дистрибутива Linux подходящего для разных сфер применения.

- *Работа на устройствах с ограниченными ресурсами.* В отличие от полных дистрибутивов Linux в YP можно создать дистрибутив, содержащий лишь требуемые встраиваемому устройству компоненты. Можно ограничиться функциями и пакетами, без которых устройство не сможет работать, а все остальное исключить. Для устройств с мониторами можно использовать системные компоненты X11, GTK+, Qt, Clutter, SDL и др., обеспечивающий широкий набор функций. Для простых устройств без монитора можно выбрать простой пользовательский интерфейс без мало нужных компонент.

- *Полнофункциональный инструментарий.* Инструментарий для поддерживаемых архитектур подойдент для большинства случаев. Однако, если ваше оборудование поддерживает возможности, не доступные стандартным инструментам, это можно легко исправить, настроив инструментарий через параметры, зависящие от платформы. Если же требуется использовать сторонние инструменты, механизмы YP позволят сделать это.

- *Правила механизма взамен политики.* Сосредоточение внимания на механизмах, а не политике обеспечивает свободу выбора правил на основе потребностей вместо приспособления к решениям того или иного поставщика программ.

- *Использование уровней.* Многоуровневая инфраструктура YP группирует связанные между собой функции в отдельные наборы (уровни). Такой уровень можно при необходимости добавить непосредственно в проект. Использование уровней для разделения и группировки функций снижает уровень сложности и избыточности, позволяя без большого труда расширять систему и должны образом организовать функциональность.

- *Поддержка частичной сборки.* При необходимости можно собирать и перестраивать отдельные пакеты. YP обеспечивает это с помощью схемы кэширования состояний с общим доступом (sstate). Возможность сборки и отладки отдельных компонент существенно облегчает разработку проектов.

- *Плановые выпуски.* Основные выпуски системы происходят каждые 6 месяцев, приблизительно в октябре и апреле. Для двух последних выпусков поддерживаются точечные обновления для устранения обнаруженных дефектов и уязвимостей. Такая предсказуемость важна для проектов на основе YP и позволяет разработчикам планировать свои действия.

- *Доступ в сообщество для индивидуалов и организаций.* Для проектов с открытым кодом сообщество играет очень важную роль. Доступны форумы поддержки, опыт и активные разработчики, развивающие YP.

- *Воспроизводимость на двоичном уровне.* YP позволяет точно указать зависимости и обеспечить очень высокий процент воспроизводимости на двоичном уровне (например, 99,8% для core-image-minimal). Если дистрибутив не задаёт извлекаемые пакеты и их порядок для поддержки зависимостей, другие системы сборки могут произвольно включать пакеты.

- *Манифест лицензирования.* YP поддерживает манифест лицензий для просмотра людьми, которым нужно контролировать использование лицензий программ с открытым кодом (например, юристы компании).

2.1.2. Сложности

Ниже рассмотрены некоторые сложности, которые могут возникать при разработке с использованием YP.

- *Скорость обучения.* YP требует изучения большого объёма материала и имеет множество способов решения похожих задач. Это может вызывать трудности при выборе конкретного варианта из множества возможных.

- *Для понимания вносимых изменений могут потребоваться некоторые исследования.* Помимо простого обучения для пониманию изменений, которые нужно внести в конкретный проект могут потребоваться значительные исследования. Краткую информацию, которая поможет перейти от пробного использования YP к разработке реальных проектов можно найти на сайте YP в документах What I wish I'd Known и Transitioning to a Custom Environment for Systems Development.

- *Рабочий процесс проекта может приводить к путанице.* Рабочий процесс YP может оказаться запутанным, если его использовать для разработки традиционный серверных и пользовательских программ. В среде разработки пользовательских систем имеются механизмы для простого извлечения и установки новых пакетов, которые обычно содержат заранее собранные двоичные файлы, доступные на серверах Internet. При работе с YP нужно самостоятельно менять конфигурацию и добавлять дополнительные пакеты.

- *Работа в среде кросс-сборки может представиться неудобной.* При разработке кода для работы на целевой платформе компиляция, исполнение и тестирование на реальной платформе могут выполняться быстрее, чем сборка BitBake на хосте разработки и последующее развёртывание двоичных файлов на целевой платформе. Хотя YP поддерживает средства разработки на целевой системе, потребуются дополнительные шаги для переноса изменений в среду сборки YP. В YP поддерживается промежуточный вариант, включающий внесение изменений на хосте разработки в среде BitBake и развёртывание на целевой системе только обновлений.

  Система сборки YP OE поддерживает стандартные форматы пакетов (RPM, DEB, IPK, TAR). Эти пакеты можно развернуть на работающей целевой системе с использованием предоставляемых там утилит, таких как rpm.

- *Время первоначальной сборки может быть продолжительным.* К сожалению долгой первоначальной сборки избежать невозможно по причине большого числа пакетов, собираемых изначально для создания полнофункциональной системы Linux. Однако после этой сборки механизм кэширования общего состояния (sstate) позволяет YP значительно ускорить последующую сборку изменённых пакетов.

2.2. Модель уровней YP

Модель уровней YP является моделью разработки для создания дистрибутивов Linux встраиваемых систем и IoT, которая отличает YP от других простых систем сборки. Модель уровней поддерживает совместную работу и настройку конфигурации уровней. Уровни являются репозиториями, содержащими наборы инструкций, которые говорят системе сборки OE, что нужно делать. Уровни можно использовать многократно, обобщать и применять в совместной работе.

Уровни могут содержать изменения созданных ранее настроек или инструкций. Мощные средства переопределения позволяют настраивать созданные ранее коллегами или сообществом уровни в соответствии со своими требованиями.

Можно применять разные уровни для логического разделения информации в вашей сборке. Например, можно создать уровни для BSP, GUI, настройки дистрибутива, промежуточных программ (middleware) или пользовательских приложений. Включение сборки целиком в один уровень ограничивает и усложняет последующую настройку и повторное использование, а разделение информации по уровням помогает упросить это.

- По возможности следует применять уровни BSP.
- Следует ознакомиться со списком уровней, поддерживаемых Yocto Project или OE (здесь больше уровней, но они менее тщательно проверяются).
- Уровни поддерживают включение технологий, аппаратных и программных компонент. Маркировка YP Compatible обеспечивает подтверждение минимального уровня стандартизации. YP Compatible применяется к продукции и программным компонентам, таким как BSP, совместимые с OE уровни и проекты с открытым кодом, что позволяет производителю использовать знаки активы YP.

Рассмотрим применение уровней на примере настройки конфигурации машины. Такие настройки обычно помещаются на отдельный уровень, называемый уровнем BSP. Кроме того, изолируем настройки машины от заданий и метаданных, которые поддерживают, например, новую среду GUI. Это разумней всего сделать на двух уровнях - один для конфигурации машины, другой для среды GUI. Важно понимать, что уровень BSP может вносить машинозависимые дополнения в среду GUI, не внося изменений непосредственно в уровень GUI. Это делается с помощью файлов дополнения BitBake (.bbappend), описанных ниже. Общая информация о структуре уровня BSP приведена в [4].

Каталог источников содержит базовые уровни и уровни BSP. Уровни из состава выпуска YP легко найти в каталоге источников по их именам, которые обычно начинаются с префикса meta-. Наличие префикса не требуется, но принято сообществом YP как стандартная практика.

Если посмотреть представление дерева репозитория poky, можно увидеть уровни meta, meta-skeleton, meta-selftest, meta-poky и meta-yocto-bsp. Процедуры создания уровней описаны в разделе Understanding and Creating Layers [1].

2.3. Компоненты и инструменты

YP обеспечивает набор компонент и инструментов, используемых самим проектом и разработчиками других проектов. Эти компоненты и инструментарий представляют собой проекты с открытым кодом и метаданные, разделённые на эталонный дистрибутив Poky и систему сборки OE. Многие компоненты и инструменты можно загружать независимо.

2.3.1. Средства разработки

Ниже кратко описаны инструменты для разработки образов и приложений с использованием YP.

- CROPS1 - открытая среда кроссплатформенной разработки, использующая контейнеры Docker. CROPS предоставляет легко управляемую и расширяемую среду сборки программ для различной архитектуры, включая хосты Windows, Linux и Mac OS X.
- *devtool* - инструмент с консольным интерфейсом, являющийся основной частью расширяемого SDK (eSDK[1]). Можно применять devtool для сборки, тестирования и создания пакетов программ в рамках eSDK. Инструмент можно использовать для интеграции сборки в образ, создаваемый системой сборки OE.

  Инструмент devtool поддерживает множество команд, которые позволяют добавлять, изменять и обновлять задания. Как и в системе сборки OE, задание в devtool представляет программный пакет. Команда devtool add позволяет создать задание автоматически. Команда devtool modify использует указанное имеющееся задание для определения источника исходных кодов и способа наложения изменений (patch). В обоих случаях среда настраивается так, чтобы при сборке задания использовалось дерево исходных кодов, находящееся под вашим контролем, что позволяет вносить в код изменения. По умолчанию новые задания и исходный код для них помещаются в каталог workspace файловой структуры eSDK. Команда devtool upgrade обновляет имеющееся задание с учётом внесённых в исходный код изменений. Работа с devtool подробно описана в разделе Using devtool in Your SDK Workflow [2].
- *Расширяемый комплект для разработки программ (eSDK)* обеспечивает инструменты и библиотеки для кросс-разработки, адаптированные для создания конкретного образа. Пакеты eSDK позволяют легко добавлять приложения и библиотеки в образ, менять исходный код имеющихся компонент, тестировать изменения на целевой платформе и интегрировать в систему сборки OE. Инструменты eSDK с набором команд devtool адаптированы для среды YP. Описание eSDK приведено в [2].
- *Toaster* - web-интерфейс для системы сборки YP OE, позволяющий настраивать и запускать сборку, а также просматривать её результаты. Описание Toaster приведено в [6].

2.3.2. Инструменты для повышения производительности работы

- *Auto Upgrade Helper* - утилита, применяемая с системой сборки OE (BitBake и OE-Core) для автоматического заданий на основе новой версии опубликованных исходных кодов.

[1]Extensible Software Development Kit - расширяемый комплект для разработки программ.

- *Recipe Reporting System* отслеживает версии заданий, доступные для YP. Основной задачей утилиты является помощь в поддержке заданий и динамическое рецензирование проекта. Система работает на основе web-сайта OpenEmbedded Layer Index, где индексируются уровни OpenEmbedded-Core.

- *Patchwork* является ветвью проекта OzLabs и представляет собой web-систему отслеживания для оптимизации процесса внесения вкладов в проект. YP использует Patchwork для обслуживания правок (patch), выпускаемых в большом количестве для каждого выпуска.

- *AutoBuilder* служит для автоматизации тестирования сборки и контроля качества (QA[1]). С помощью общедоступного сервиса AutoBuilder каждый может определить статус текущей ветви master в Poky. AutoBuilder работает на основе buildbot.

  YP стремится быть передовым проектом в сфере открытого кода для автоматизации тестирования и проверки качества (QA). Проект приветствует публикацию разработчиками планов QA и тестирования, открыто демонстрирует свои планы и призывает разрабатывать инструменты для автоматизации процедур тестирования и QA на благо сообщества. Информацию о работе с AutoBuilder в YP можно найти по ссылке.

- *Cross-prelink.* Предварительной ссылкой называется процесс упреждающего вычисления адресов загрузки и таблиц связей (компоновки) динамическим компоновщиком. Это позволяет повысить скорость запуска приложений и снизить расход памяти общими для множества приложений библиотеками.

  Исторически предварительные кросс-ссылки являются вариантом предварительных ссылок, задуманных Jakub Jelínek много лет назад. Те и другие ссылки поддерживаются в разных ветвях одного репозитория. За счёт предоставления эмулируемого динамического компоновщика в процессе работы (эмуляция ld.so из glibc) проект cross-prelink расширяет возможности программ с упреждающими ссылками, позволяя работать в средах стиля sysroot.

  Динамический компоновщик вычисляет стандартные адреса загрузки на основе таких факторов, как адреса отображения, использование библиотек и конфликты библиотечных функций. Инструмент prelink использует эту информацию от динамического компоновщика для определения уникальных адресов загрузки двоичных файлов ELF[2], являющихся динамическими библиотеками. Инструмент prelink меняет эти файлы ELF, используя заранее рассчитанные данные. Результатом является ускорение загрузки и зачастую экономия памяти, поскольку большая часть кода библиотеки может использоваться многократно со страниц COW[3].

  Исходный проект prelink поддерживает работу prelink лишь на динамических компоновщиках конечных целевых устройств. Это ограничение вызывает проблемы при разработке систем с кросс-компиляцией. Проект cross-prelink добавляет синтезированный динамический загрузчик, работающий на хосте и позволяющий создавать предварительные кросс-ссылки без запуска на целевой платформе с возможностью чтения и записи.

- *Pseudo* - реализация в YP системы fakeroot, используемой для запуска команд в среде, которая представляется имеющей привилегии root.

  В процессе сборки может потребоваться выполнение операций с полномочиями системного администратора. Например, может потребоваться установка принадлежности (владения) и прав доступа для файлов. Инструмент Pseudo можно использовать напрямую или через переменную среды LD_PRELOAD. В любом случае это позволяет выполнять операции с правами системного администратора, даже если его нет. Информация об инструменте приведена в параграфе 4.7. Fakeroot и Pseudo.

2.3.3. Компоненты системы сборки OE

- *BitBake* является основной компонентой YP и применяется системой сборки OE для создания образов. Несмотря на это, BitBake поддерживается отдельно от YP.

  BitBake является базовой машиной выполнения задач, обеспечивающей эффективное выполнение задач командного процессора и Python с возможностью параллельной работы с учётом зависимостей между задачами. Кратко говоря, BitBake - это машина сборки, работающая с заданиями в определённом формате для выполнения набора задач. Описание BitBake приведено в [7].

- *OpenEmbedded-Core* (OE-Core) - базовый уровень метаданных (задания, классы и связанные с ними файлы), используемых основанными на OE системами, включая YP. Проекты YP и OpenEmbedded поддерживают OpenEmbedded-Core. Метаданные OE-Core доступны в репозитории YP Source Repositories.

  Исторически метаданные OE-Core были интегрированы в YP через репозиторий эталонного дистрибутива Poky. После YP версии 1.0 было принято решение о совместной работе и использовании в YP и OE общего базового набора метаданных (OE-Core), который обеспечивает функциональность, ранее наблюдавшуюся в Poky. Это сотрудничество позволило достичь долгосрочной цели OE в части получения более строго контролируемого ядра с гарантией качества. Результат также соответствует цели YP получить меньшее, чем в иных системах число полнофункциональных инструментов.

  Совместное использование ядра метаданных сделало Poky интеграционным уровнем на основе OE-Core, как показано на рисунке в разделе 2.1. Что такое YP. YP объединяет такие компоненты, как BitBake, OE-Core, сценарий «склеивания» и документацию по системе сборки.

2.3.4. Эталонный дистрибутив Poky

Poky является эталонным дистрибутивом YP и содержит систему сборки OpenEmbedded (BitBake и OE-Core), а также набор метаданных, позволяющие начать создание своего дистрибутива. Рисунок в разделе 2.1. Что такое YP показывает связь Poky с другими частями YP.

Для использования инструментов и компонент YP можно загрузить (клонировать) Poky и использовать его в качестве исходной точки для создания своего дистрибутива. Poky не включает двоичных файлов и является примером создания

[1]Quality assurance - гарантии качества.
[2]Executable and linkable format - исполняемый и компонуемый формат.
[3]Copy-On-Write - копирование при записи.

дистрибутива Linux на базе исходных кодов. Дополнительные сведения о Poky приведены в разделе 2.5. Эталонный дистрибутив Poky.

2.3.5. Готовые пакеты

- *Matchbox* - открытая базовая среда X Window для работы на встраиваемых системах, таких как карманные компьютеры, телевизионные приставки, киоски и т. п. с ограниченными размером экрана, механизмами ввода и системными ресурсами. Matchbox включает множество взаимозаменяемых и необязательных приложений, которые можно приспособить к конкретной платформе для расширения её функциональности. Исходные коды Matchbox доступные в репозитории YP Source.

- *Opkg*[1] - облегчённая система управления пакетами на основе itsy (ipkg), написанная на языке C и похожая в работе на Advanced Package Tool (APT) и Debian Package (dpkg). Менеджер пакетов предназначен для встраиваемых систем на основе Linux и применяется в проектах OpenEmbedded, OpenWrt и YP. Насколько это возможно, opkg поддерживает совместимость с ipkg и частично соответствует правилам Debian к файлам управления.

2.3.6. Архивные компоненты

Build Appliance - это образ виртуальной машины, позволяющий собрать и загрузить свой образ Linux для встраиваемой системы с помощью YP на платформе разработки, отличной от Linux. Исторически Build Appliance был вторым из 3 методов использования YP в системе, которая отличается от Linux.

1. *Hob* в настоящее время считается устаревшим и недоступен, поскольку с выпуска 2.1 YP поддерживает элементарный интерфейс GUI (Toaster), полностью заменивший Hob.

2. *Build Appliance* стал доступным после Hob, но никогда не был рекомендован в качестве повседневной среды разработки в YP. Build Appliance служит полезным вариантом пробной разработки в среде YP.

3. *CROPS* является финальным и наиболее эффективным решением для разработки с использованием YP на системах, отличных от Linux.

2.4. Методы разработки

Среда разработки YP обычно включает сборочный хост и целевую аппаратную платформу. Хост служит для подготовки образов и разработки приложений, которые разворачиваются на целевой платформе для тестирования.

В этом разделе кратко описаны методы разработки, выбираемые при настройке хоста сборки. В зависимости от конкретных задач и операционной системы хоста сборки возможны несколько вариантов использования YP. Глава 3. Среда разработки YP содержит дополнительные сведения о среде разработки YP.

- *Хост Linux* на сегодняшний день обеспечивает наилучшее решение для сборки. Linux является естественной операционной системой, позволяющей напрямую разрабатывать программы с использованием инструментов BitBake. Вся разработка может выполняться в привычной среде поддерживаемого дистрибутива Linux. Информация об организации хоста сборки в среде Linux приведена в разделе Setting Up a Native Linux Host [1].

- *CROss PlatformS (CROPS)* на основе контейнеров Docker обычно применяется для организации хоста сборки под управлением других ОС (например, Microsoft® Windows™ или macOS®), однако можно использовать CROPS и в среде Linux.

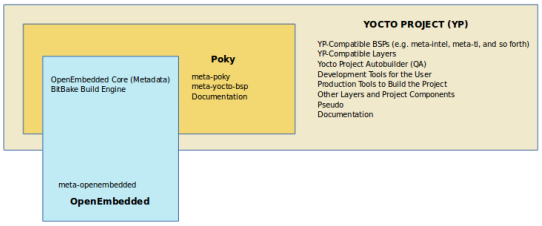
  CROPS является открытой системой кроссплатформенной разработки, которая обеспечивает управляемую и расширяемую среду для сборки двоичных файлов, предназначенных для разной архитектуры на хостах Windows, macOS или Linux. После настройки хоста сборки CROPS можно организовать среду разработки, имитирующую Linux. Информация о настройке хоста сборки CROPS дана в разделе Setting Up to Use CROss PlatformS (CROPS) [1].

- *Toaster* позволяет разрабатывать программы с использованием YP, независимо от ОС хоста сборки. Toaster обеспечивает web-интерфейс для системы сборки YP OpenEmbedded, позволяющий настраивать и запускать процесс сборки. Информация о сборке собирается и хранится в базе данных. Toaster можно применять для настройки и запуска сборки на нескольких удалённых серверах сборки. Описание Toaster приведено в [6].

2.5. Эталонный дистрибутив Poky

Poky (произносится *Pock*-ее) — эталонный дистрибутив YP или Reference OS Kit. Poky включает систему сборки OE (BitBake и OpenEmbedded-Core), а также набор метаданных для построения своего дистрибутива. Т. е. Poky задаёт базовую функциональность, требуемую для типовых встраиваемых систем, а также компоненты YP, позволяющие создать полезные двоичные образы. Poky объединяет репозитории BitBake, OpenEmbedded-Core (каталог meta), meta-poky, meta-yocto-bsp и документацию. Все эти компоненты Poky представлены в репозитории Source.

Полностью содержимое репозитория poky Git представлено в разделе Top-Level Core Components [3].



На рисунке представлено содержимое Poky.

- BitBake - менеджер и планировщик задач, являющийся центральной частью системы сборки OE.

[1]Open PacKaGe management - открытая система управления пакетами.

- meta-poky содержит относящиеся к Poky метаданные.
- meta-yocto-bsp относится к пакетам поддержки плат в YP (BSP).
- Метаданные OpenEmbedded-Core (OE-Core) содержат конфигурации, определения глобальных переменных, классы общего пользования, средства подготовки пакетов и задания. Классы определяют инкапсуляцию и наследование логики сборки, задания являются логическими блоками собираемых программ и образов.
- Документация содержит исходные файлы YP для создания руководств пользователя.

Хотя Poky задаёт полную спецификацию дистрибутива и проверен с помощью QA, он не является «готовой продукцией». Для работы с инструментами YP можно клонировать репозиторий Poky с помощью Git и далее пользоваться локальной копией для создания своего дистрибутива. Poky не включает готовых двоичных файлов и является лишь примером создания дистрибутива Linux из исходных кодов.

Poky выпускается каждые 6 месяцев со своей нумерацией версий. Основные выпуски выходят синхронно с выпусками YP, обычно весной и осенью. Дополнительная информация о планировании выпусков YP приведена в разделе Yocto Project Releases and the Stable Release Process [3].

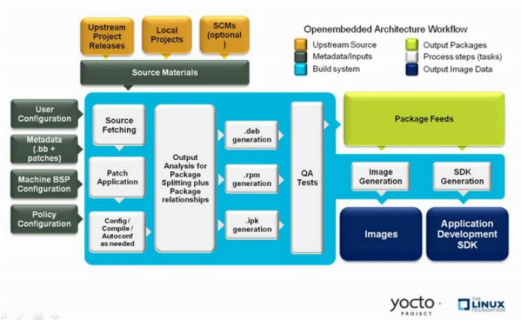
Выше уже было сказано, что Poky является «принятой по умолчанию конфигурацией», которая служит исходной точкой для создания образа. Можно использовать Poky напрямую, создавая образы от самого простого до соответствующего требованиям Linux Standard Base (LSB) с эталонным пользовательским интерфейсом GNOME Mobile and Embedded (GMAE), который называется Sato.

Одним из важных свойств Poky является управление всеми аспектами сборки через метаданные. Можно расширить базовый образ, добавив уровни метаданных с расширенными функциями. Эти уровни могут добавлять программный стек в тип образа, добавлять пакеты поддержки плат (BSP) для оборудования, создавать новые типы образов.

Метаданные свободно группируются в конфигурационные файлы или задания для пакетов. Задание представляет собой набор метаданных, используемых BitBake для определения переменных или дополнительных задач при сборке. Файл задания содержит описание и версию задания, лицензию пакета и ссылку на репозиторий исходных кодов. Задание может также указывать процесс сборки с использованием autotools, make, distutils и т. п., при этом базовая функциональность будет определяться классами, наследуемыми из определений уровня OE-Core в ./meta/classes. В задании могут также указываться дополнительные задачи в качестве предварительных условий. Синтаксис заданий также поддерживает операторы _prepend и _append для расширения функциональности задач. Эти операторы добавляют код в начале или в конце задачи. Описание операторов этих BitBake приведено в разделе Appending and Prepending (Override Style Syntax) [7].

2.6. Работа с системой сборки OE

Система сборки OE использует для создания образов и SDK рабочий процесс (workflow), показанный на рисунке.



1. Разработчики задают архитектуру, правила, правки (patch) и конфигурацию.
2. Система сборки извлекает исходные коды из указанных мест. Поддерживаются стандартные методы, включая архивы (tarball) и репозитории систем управления исходным кодом, таких как Git.
3. После загрузки исходных кодов система сборки распаковывает архивы в локальную рабочую область, где применяются исправления и общие операции по настройке конфигурации и компиляции программ.
4. Система сборки устанавливает программы во временную область, где применяется выбранный формат (DEB, RPM, IPK) подготовки пакетов.
5. Выполняются процедуры QA и проверки работоспособности на протяжении всего процесса сборки.
6. После создания двоичных файлов система сборки создает хранилище двоичных пакетов для создания окончательного образа корневой файловой системы.
7. Система сборки создает образ файловой системы и eSDK.

Более подробное описание рабочего процесса приведено в разделе 4.3. Концепции системы сборки OE.

2.7. Основные термины

Список используемых в YP приведен в разделе Yocto Project Terms [3], а здесь даны определения базовых терминов, которые помогут при изучении YP.

- *Файлы конфигурации* содержат определения глобальных и пользовательских переменных, а также данные конфигурации оборудования. Эти файлы говорят системе сборки OE, что нужно собрать и что включить в образ для поддержки конкретной платформы.
- *Расширяемый комплект для разработки программ (eSDK)* - это пользовательский SDK для разработчиков приложений, позволяющий встраивать свои изменения программ и библиотек в образ для использования другими разработчиками. Описание eSDK приведено в [2].

- *Уровень (Layer)* - это набор связанных между собой заданий. Уровни позволяют объединить связанные между собой метаданные для настройки сборки, а также разделить информацию при сборках для разной архитектуры. Уровни являются иерархическими и могут переопределять спецификации других уровней. Можно взять любое число уровней из YP (см. список уровней на сайте) и настроить сборку, добавив свои уровни.

  Подробное описание уровней приведено в разделе Understanding and Creating Layers [1], а уровни BSP дополнительно рассмотрены в разделе BSP Layers [4].

- *Метаданные* служат важнейшим элементом YP, используемым для создания дистрибутива Linux, и содержатся в файлах, которые система сборки OE анализирует в процессе создания образа. В общем случае метаданные включают задания, файлы конфигурации и другую информацию, которая относится к самим инструкциям сборки, а также данные для выбора цели сборки. Метаданные включают также команды и данные, используемые для указания используемых версий программ, источников кода, применяемых исправлений и дополнений, которые служат для исправления ошибок или настройки программ под конкретную ситуацию. Важный для работы базовый набор проверенных метаданных содержится в OpenEmbedded-Core.

- *Система сборки OE,* которую иногда называют BitBake или просто «система сборки».

  BitBake является планировщиком и машиной выполнения задач, которая обеспечивает синтаксический разбор инструкций (задания) и данных конфигурации, создавая дерево зависимостей для компиляции, планирует компиляцию включённого кода, а затем выполняет сборку пользовательского образа Linux (дистрибутива). BitBake по своей функциональности напоминает make.

  В процессе сборки отслеживаются зависимости и выполняется естественная (native) или кросс-компиляция пакетов. В качестве первого шага кросс-сборки предпринимается попытка создания набора инструментов кросс-компиляции (eSDK), подходящего для целевой платформы.

- *OpenEmbedded-Core (OE-Core)* - это набор метаданных, включающий задания, классы и связанные с ними файлы, которые являются общими для множества разных систем, основанных на OE, включая YP. OE-Core представляет собой подмножество репозитория, разработанного сообществом OE, включающее наиболее часто применяемые задания. Результатом сокращения стал строго контролируемый набор заданий с гарантированным качеством. Метаданные доступны в каталоге meta репозитория исходных кодов YP.

- *Пакеты.* В контексте YP пакетами называют задания, создаваемые BitBake (выполненные задания). Пакет обычно содержит двоичные файлы, скомпилированные из исходных кодов задания.

  Следует отметить, что трактовка термина «пакет» может включать некоторые нюансы. Например, пакеты, упоминаемые в разделе Required Packages for the Build Host [3], являются скомпилированными двоичными файлами, установка которых расширяет функциональность дистрибутива Linux.

  Следует также отметить, что исторически задания в YP назывались пакетами, поэтому имена некоторых переменных BitBake (например PR, PV, PE) не совсем корректны.

- *Poky* является эталонным дистрибутивом для встраиваемых систем и эталонной тестовой конфигурацией. Poky включает в себя:

  - дистрибутив с базовой функциональностью, служащий для иллюстрации настройки дистрибутивов;
  - средства тестирования компонент YP (Poky применяется для проверки пригодности YP);
  - средства загрузки YP.

  Poky не является законченным дистрибутивом и служит лишь отправной точкой для создания дистрибутива. Poky представляет собой интеграционный уровень на основе OE-Core.

- *Задание (Recipe)* представляет собой наиболее распространённую форму метаданных и содержит список настроек и задач (инструкций) для сборки пакетов, которые включаются в образ. Задание описывает местоположение исходного кода и применение к нему исправлений (patch), а также зависимости от библиотек или других заданий, конфигурацию и опции компиляции. Связанные между собой заданий образуют уровень.

Глава 3. Среда разработки YP

В этой главе описана среда разработки YP с рассмотрением концепций YP, что поможет разобраться с работой в среде с открытым кодом, которая существенно отличается от работы в закрытых фирменных средах. В частности, здесь описана философия программ с открытым кодом, репозитории кода, рабочие процессы, Git и лицензирование.

3.1. Философия программ с открытым кодом

Философия открытого кода характеризуется разработкой программ, ориентированной на совместную работу и сотрудничество сообщества активных разработчиков. Это отличается от более стандартизованных моделей централизованной разработки, используемых коммерческими компаниями, когда создаваемая для продажи продукция, разрабатывается конечным набором людей с использованием определённого набора процедур и результатом служит закрытая от сообщества система для определённой архитектуры и платформы.

Проекты с открытым кодом имеют множество параллельных задач, подходов и результатов. Детали разработки могут исходить от любого человека (сообщество), который заинтересован участвовать в проекте. В среде с открытым кодом вопросы авторских прав, лицензирования, сферы распространения и потребителей решаются иначе, нежели в традиционных средах разработки. Здесь конечный результат, исходные коды и документация доступны бесплатно всем желающим.

Типичным примером проекта с открытым кодом является ядро Linux, которое было задумано и создано финским студентом Линусом Торвальдсом (Linus Torvalds) в 1991 г. А примером закрытого кода может служить семейство операционных систем Windows® от Microsoft® Corporation.

В Wikipedia приведено хорошее историческое описание философии открытого кода.

3.2. Хост разработки

Для работы с YP требуется хост разработки или сборочный хост. Поскольку целью YP является создание образов и приложений для работы на встраиваемых системах, процесс разработки и сборки происходит не на той системе, где будет применяться образ или программа, а на хосте разработки. Большинство людей считает что в качестве сборочного хоста лучше всего использовать машину с операционной системой Linux, однако можно работать и с системами Mac или Windows, применяя CROPS на базе контейнеров Docker. После организации машины CROPS вы получите доступ к среде, похожей на среду хоста разработки Linux. Организация машины CROPS описана в разделе Setting Up to Use CROss PlatformS (CROPS) [1].

На хосте разработки Linux также требуется ряд операций для подготовки к работе с YP. Дистрибутив Linux должен поддерживать YP. Кроме того, потребуется установить на хосте ряд пакетов, требуемых для разработки с использованием YP. Процедуры установки описаны в разделе Setting Up a Native Linux Host [1].

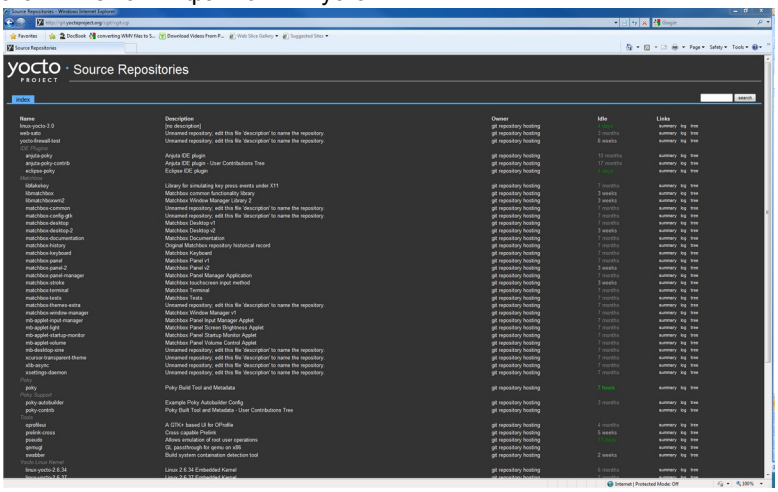
После организации хоста разработки YP можно пользоваться несколькими вариантами работы, описанными ниже.

- *Командная среда BitBake.* Традиционный процесс разработки в YP включает использование системы сборки OE, которая включает командный интерфейс BitBake на хосте разработки. Интерфейс доступен как в среде Linux, так и на машинах CROPS. В обоих случаях образы создаются, изменяются и собираются в командной среде, использующей компоненты и инструменты, доступные в дистрибутиве Linux и YP. Общее описание процедур сборки приведено в разделе Building a Simple Image [1].
- *Разработка BSP* включает использование YP для создания и тестирования уровней, позволяющих разрабатывать образы и приложения для конкретного оборудования. Для разработки BSP требуются дополнительные операции по организации хоста сборки, описанные в разделе Preparing Your Build Host to Work With BSP Layers [4].
- *Разработка ядра* с использованием YP чаще всего выполняется с помощью инструмента devtool, существенно упрощающего работу и ускоряющего процесс. Подготовка хоста разработки к работе с ядром описана в разделе Preparing the Build Host to Work on the Kernel [9].
- *Интерфейс Toaster* системы сборки OE обеспечивает средства для настройки и запуска сборки. Информация о сборке сохраняется в базе данных. Интерфейс Toaster позволяет настраивать и запускать сборку на нескольких удалённых сборочных хостах. Установка, настройка и использование Toaster описаны в [6].

3.3. Репозитории исходных кодов YP

Команда YP поддерживает репозитории всех файлов на сайте http://git.yoctoproject.org. Репозитории организованы по категориям, таким как IDE Plugins, Matchbox, Poky, Yocto Linux Kernel и т. п. Можно выбрать любой элемент в колонке Name и увидеть внизу страницы ссылку URL для клонирования репозитория Git с выбранным элементом. Наличие локальной копии репозитория Git в каталоге исходных кодов, которые обычно именуется poky, позволяет вносить изменения, обеспечивая в конечном итоге развитие инструментов YP, BSP и т. п..

Для любого из поддерживаемых выпусков YP можно перейти на сайт Yocto Project и выбрать вариант DOWNLOADS в меню SOFTWARE для загрузки архива репозитория repository, поддерживаемых BSP и инструментов YP. Распаковка этих архивов обеспечит локальные копии файлов выпуска.



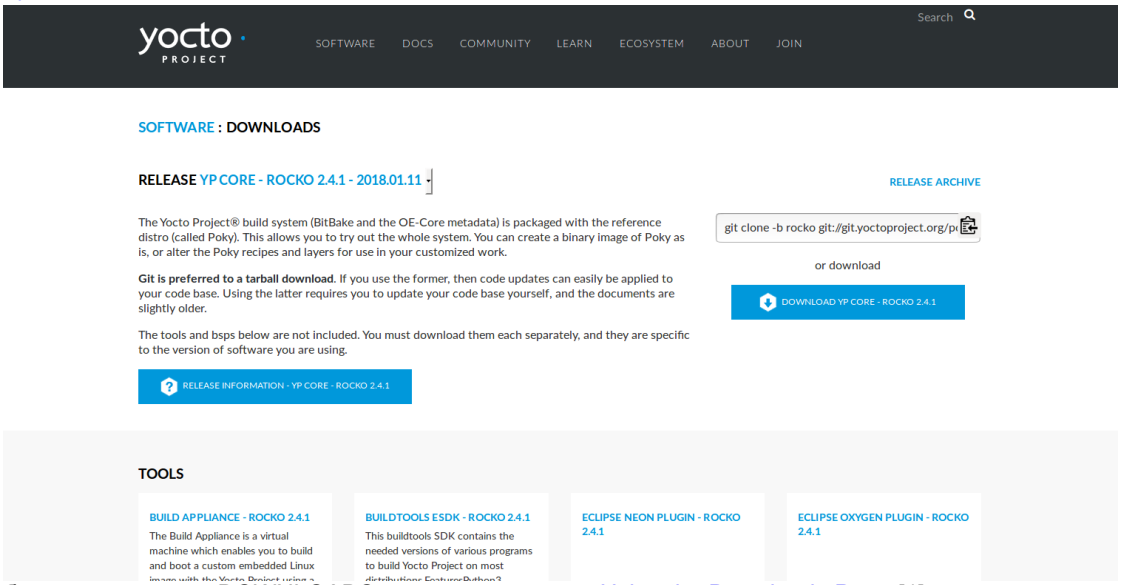
- Для установки каталога источников YP и файлов поддерживаемых BSP (например, meta-intel) рекомендуется использовать Git.
- Следует всегда применять соответствующие ветви репозитория выбранного BSP и исходного кода (poky). Например, при выборе ветви master для poky и работе с meta-intel нужно выбрать ветвь master в meta-intel.

Ниже кратко рассмотрены источники получения файлов, требуемых для разработки.

- *Репозиторий исходных кодов* включает категории IDE Plugins, Matchbox, Poky, Poky Support, Tools, Yocto Linux Kernel и Yocto Metadata Layer. Можно создать локальные копии каждого из репозиториев Git. Работа с репозиториями Git описана в разделе Accessing Source Repositories [1].
- *Список выпусков* содержит Poky, Pseudo, установщики инструментов кросс-разработки, а также компоненты поддержки и все выпуски YP в форме образов или архивов. Загрузка и распаковка этих файлов создает не локальную копию репозитория Git, а «снимок» определённого выпуска или образа.

Работа с такими файлами описана в разделе Accessing Index of Releases [1].

- *Страница DOWNLOADS на сайте Yocto Project Website*, доступная через меню SOFTWARE, позволяет загрузить любой выпуск YP, инструменты и BSP в виде архивов (tarball), подобных представленным в списке выпусков.



Работа со страницей DOWNLOADS описана в разделе Using the Downloads Page [1].

3.4. Работа с Git в YP

Разработка с использованием YP требует применения Git - открытой распределенной системы управления исходными кодами,которая применяется во многих средах коллективной разработки. В этом разделе описан процесс работы с YP и Git, в частности, описаны базовые методы, роли и операции в среде коллективной разработки.

Файлы YP поддерживаются с использованием Git в ветвях (branch) и Git отслеживает каждое изменение и структуру ветвления. Хотя применение Git для этого не обязательно, многие открытые проекты используют такой подход.

В YP сопровождающий (maintainer) отвечает за целостность ветви master данного репозитория Git. Ветвь master является «восходящим» (upstream) репозиторием, на основе которого выполняется сборка для текущего состояния проекта. Сопровождающий отвечает за принятие изменений от других разработчиков и организацию базовой структуры ветви в соответствии со стратегией выпусков и т. п. Способы выяснения сопровождающих, которые отвечают за конкретную ветвь (и поддерживают её) описаны в разделе Submitting a Change to the Yocto Project [1].

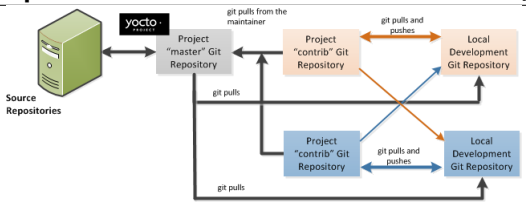
Репозиторий YP poky включает также «восходящий» репозиторий Git с именем poky-contrib. Можно увидеть все ветви этого репозитория через web-интерфейс Source Repositories в области Poky Support. В этих ветвях содержатся изменения (commit) в проекте, которые были представлены или зафиксированы командой разработки YP и членами сообщества, участвующими в проекте. Сопровождающий определяет пригодность изменений для переноса из ветвей contrib в ветвь master репозитория Git.

Разработчики (включая членов сообщества) создают и поддерживают клонированные репозитории разрабатываемых ветвей, которые размещаются локально на их платформах и служат для подготовки изменений. Когда разработчик удовлетворён полученными результатами, он «выталкивает» (push) эти изменения в подходящий репозиторий contrib.

Разработчики сами отвечают за поддержку актуальности локального репозитория, независимо от ветви, с которой они работают. Они также отвечают за устранение конфликтов, которые могут возникать в результате работы с файлами нескольких человек сразу. Вся эта работа выполняется локально на хосте разработки до выталкивания в область contrib, где её проверяет сопровождающий.

Имеется несколько формализованных методов, с помощью которых разработчики фиксируют изменения и помещают их в область contrib, а затем просят сопровождающего включить их в разрабатываемую ветвь. Этот процесс называется представлением изменений (submitting a patch или submitting a change). Процедуры описаны в разделе Submitting a Change to the Yocto Project [1].

Таким образом организуется «единая точка входа» в ветвь master или разрабатываемую ветвь репозитория Git, контролируемая сопровождающим проект лицом. Разработчики независимо создают, тестируют и представляют изменения в области contrib для проверки сопровождающим, который выбирает включаемые в основной проект правки.



Несмотря на уникальность каждой среды разработки, имеются общие методы «бесшовной» стыковки результатов, которые кратко описаны ниже. Более полное описание процессов приведено в [10].

- *Внесение изменений небольшой группой.* Лучше вносить изменения небольшими порциями, нежели собирать вместе множество разнородных изменений. Это не только позволяет управлять изменениями, но и упрощает сопровождающему решение вопросов о принятии или отклонении правок.
- *Завершённые наборы изменений.* Хорошим тоном является поддержка состояния репозитория, позволяющего в любой момент собрать проект. Иными словами, не следует фиксировать незавершённые изменения. Каждая фиксация должна обеспечивать возможность сборки проекта.
- *Свободное использование ветвей.* При работе с локальным репозиторием Git легко создать, использовать и затем удалить ветвь. Локальным ветвям можно давать произвольные имена, например, связывая их со свойством или изменением, которое задаёт ветвь. После слияния изменения с основной локальной ветвью временную ветвь можно просто удалить.
- *Слияние изменений.* Команда git merge позволяет принять изменения из одной ветви в другую. Это особенно полезно при работе нескольких человек над разными частями одной функции или изменения. При слиянии автоматически обнаруживаются любые конфликты, которые могут возникать в результате изменения одной строки несколькими участниками работы.
- *Управление ветвями.* Поскольку работа с ветвями проста, следует применять ветви для указания различных уровней готовности кода. Например, можно сделать ветвь work для разработки, test - для тестирования и изменения кода по результатам тестов, stage - для кода, готового к представлению и д. д. По мере развития проекта код из разных ветвей может сливаться.
- *Сценарии Push и Pull.* Процесс «тяни-толкай» (push-pull) основан на концепции разработчиков, «вталкивающих» (push) свои локальные фиксации в удалённый репозиторий. В процессе также участвуют разработчики, «вытягивающие» (pull) известные состояния проекта в свой локальный репозиторий разработки. Процесс позволяет загрузить фиксации других разработчиков из восходящего репозитория, обеспечивая наличие самых свежих версий. YP включает сценарии create-pull-request и send-pull-request для использования этого процесса. Сценарии размещаются в каталоге scripts дерева источников. Работа со сценариями описана в разделе Using Scripts to Push a Change Upstream and Request a Pull [1].
- *Процесс исправления (Patch Workflow)* позволяет уведомить сопровождающего по электронной почте о наличии изменений (правок — patch), которые вы считаете нужным включить в ветвь master репозитория Git. Для отправки изменений формируется patch-файл и передаётся по электронной почте с помощью команд git format-patch и git send-email. Информация о работе с этими сценариями приведена в разделе Submitting a Change to the Yocto Project [1].

3.5. Git

YP широко использует Git - открытую систему управления версиями исходных кодов. Git поддерживает распределенную и нелинейную разработку и может применяться для крупных проектов. Полезно разобраться с тем, как Git отслеживает проекты и как применять Git при работе в среде YP. Ниже приведен краткий обзор работы с Git и описания некоторых важных команд Git. Более полная информация доступна на сайте Git.

При необходимости установки Git лучше сделать это через менеджер пакетов используемого дистрибутива. Доступные для загрузки файлы можно найти на странице загрузок Git. Дополнительная информация приведена в разделе Locating Yocto Project Source Files [1].

3.5.1. Репозитории, теги и ветви

Как отмечено в разделе 3.4. Работа с Git в YP, YP поддерживает репозитории исходных кодов на сайте http://git.yoctoproject.org. В web-интерфейсе видно, что каждый элемент хранится в своём репозитории Git.

Репозитории применяют ветвление Git с отслеживанием изменений содержимого (не файлов) в рамках проекта (например, добавление новых функций или обновление документации). Создание древовидной структуры на основе ветвления проекта позволяет получить полную информацию об истории всего проекта. Эта методология позволяет также использовать среду для выполнения локальных экспериментов при создании новых функций и внесении правок.

Репозиторий Git представляет все действия по разработке данного проекта. Например, репозиторий poky содержит все изменения и разработки для этого проекта за весь срок его существования. Репозиторий поддерживает полную историю изменений.

Можно создать локальную копию репозитория путём его клонирования с помощью команды git clone. Клонирование создает локальную копию репозитория в системе разработки. С этой копией можно вести разработку на своём хосте. Примеры клонирования репозиториев Git даны в разделе Locating Yocto Project Source Files [1].

Важно понимать, что Git отслеживает изменения содержимого, а не файлов. Для организации разных направлений разработки применяются ветви. Например, репозиторий poky имеет несколько ветвей, включая zeus, master и множество ветвей прошлых выпусков YP. Можно увидеть эти ветви на странице http://git.yoctoproject.org/cgit.cgi/poky/, щёлкнув ссылку [...] в разделе Branch.

Каждая из этих ветвей представляет конкретную область разработки. Ветвь master представляет текущее (или наиболее свежее) состояние основной разработки, а прочие - те или иные ответвления процесса.

При создании локального репозитория Git копия включает все ветви исходного репозитория. Это позволяет использовать Git для создания локальной рабочей области (ветви), отслеживающей ветвь разработки из восходящего репозитория Git. Иными словами, можно определять локальную среду Git для работы с любой ветвью репозитория. В качестве иллюстрации рассмотрим пример.

```
$ cd ~
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git checkout -b zeus origin/zeus
```

В этом примере после перехода в домашний каталог команда git clone создает локальную копию репозитория poky. По умолчанию Git выбирает для работы ветвь master. После перехода в каталог локального репозитория poky команда git checkout создает и выбирает локальную ветвь zeus, отслеживающую восходящую ветвь origin/zeus. Внесённые в эту ветвь изменения будут в конечном итоге влиять на ветвь zeus восходящего репозитория poky.

Важно понимать, что при создании и выборе локальной рабочей ветви по имени, локальная среда будет соответствовать «подсказке» (tip) этой конкретной ветви разработки на момент создания локальной ветви и может отличаться от ветви master в восходящем репозитории. Иными словами, создание и выбор локальной ветви по имени zeus будет отличаться от выбора ветви master в репозитории. Далее будет описано, как создать локальный снимок выпуска YP.

Git использует теги для маркировки конкретных изменений в структуре ветвей репозитория. Обычно теги служат для маркировки специальных точек, таких как финальное изменение (или фиксация) перед выпуском проекта. Можно увидеть теги репозитория poky на странице http://git.yoctoproject.org/cgit.cgi/poky/ по ссылке [...] в разделе Tag. Важными тегами poky являются jethro-14.0.3, morty-16.0.1, pyro-17.0.0 и zeus-22.0.0, представляющие выпуски YP.

В локальной копии репозитория Git сохраняется доступ ко всем тегам восходящего репозитория. Можно создавать и выбирать локальную рабочую ветвь Git по имени тега. В этом случае будет создан снимок репозитория Git, отражающий состояние файлов на момент внесения изменений, связанных с тегом. Чаще всего это применяется для выбора рабочей ветви, соответствующей определённому выпуску YP. Пример такого выбора приведен ниже.

```
$ cd ~
$ git clone git://git.yoctoproject.org/poky
$ cd poky
$ git fetch --tags
$ git checkout tags/rocko-18.0.0 -b my_rocko-18.0.0
```

Здесь каталог верхнего уровня локального репозитория YP называется poky. После перехода в этот каталог команда git fetch делает все теги восходящего репозитория доступными локально. Команда git checkout после этого создает и выбирает ветвь с именем my-rocko-18.0.0, основанную на восходящей ветви, в которой HEAD соответствует фиксации репозитория с тегом rocko-18.0.0. Файлы в локальном репозитории будут совпадать с файлами конкретного выпуска YP с указанным тегом в восходящем репозитории Git. Важно понимать, что при создании и выборе локальной ветви на основе тега среда будет соответствовать конкретному момент а не всей ветви разработки.

3.5.2. Базовые команды

Широкий набор команд Git позволяет управлять изменениями и участвовать в совместной работе над проектами. Для управления небольшим набором базовых операций и рабочих процессов следует понять базовую философию Git. Не нужно быть экспертов в Git для работы с программой. Краткий справочник по командам Git доступен по ссылке.

Ниже описаны некоторые базовые команды Git, позволяющие начать работу. Команды указаны без аргументов, полное описание доступно в документации Git.

git init
Инициализирует пустой репозиторий Git. Репозитории Git не могут использоваться без структуры .git.

git clone
Создает локальную копию репозитория Git данного разработчика или восходящего репозитория.

git add
Локально помещает обновлённое содержимое файлов в индекс, применяемый в Git для отслеживания изменений. Перед фиксацией все изменённые файлы должны быть подготовлены.

git commit
Создает локальную «фиксацию» с документированием внесённых изменений. Фиксируются лишь изменения, добавленные в индекс. Фиксации служат для сохранения истории, определения сопровождающим приемлемости изменений для проекта и в конечном итоге для переноса локальных изменений в репозиторий проекта.

git status
Сообщает о всех изменённых файлах в каталоге и индексе, указывая изменённые, но не включённые в индекс файлы и файлы, ожидающие фиксации.

git checkout branch-name
Меняет локальную рабочую ветвь в предположении её наличия. Похожа на команду оболочки cd.

git checkout –b working-branch upstream-branch
Создает и выбирает рабочую ветвь working-branch на локальной машине с отслеживанием ветви upstream-branch. Можно использовать локальную ветвь для изоляции своей работы при добавлении конкретных свойств или изменений. Изоляция ветвей упрощает удаление невостребованных изменений.

git branch
Выводит имеющиеся локальные ветви, связанные с локальным репозиторием. Текущая выбранная ветвь помечается звёздочкой.

git branch -D branch-name
Удаляет имеющуюся локальную ветвь. Для удаления локальной ветви branch-name она не должна быть выбрана.

git pull --rebase
Извлекает данные из восходящего репозитория Git и помещает их в локальный репозиторий. Команда служит для синхронизации локального репозитория с базовым (например, с ветвью master). Опция --rebase гарантирует сохранение всех локальных фиксаций при синхронизации.

git push repo-name local-branch:upstream-branch
Передаёт все зафиксированные локальные изменения в восходящий репозиторий Git, отслеживаемый локальным репозиторием. Сопровождающий просматривает эти репозитории для слияния изменений с нужной ветвью.

git merge
Объединяет или добавляет обновления из одной ветви локального репозитория с другой ветвью. При создании локального репозитория Git принятая по умолчанию ветвь называется master. Типичный рабочий процесс включает создание временной ветви на основе master, служащей для изоляции работы. Можно внести нужные изменения в эту временную ветвь, проиндексировать и зафиксировать их локально, переключиться на ветвь master и воспользоваться командой git merge для применения изменений в изолированной ветви к текущей выбранной ветви (например, master). Если временная ветвь после слияния уже не нужна, её можно удалить.

git cherry-pick commits
Выбирает и применяет указанные фиксации (commit) из одной ветви в другую. Бывают моменты, когда невозможно слить все изменения в одной ветви с другой ветвью, но нужно применить некоторые изменений.

gitk
Обеспечивает графический интерфейс (GUI) для просмотра ветвей и изменений в локальном репозитории Git. Команда даёт хорошее графическое представление изменений в локальном репозитории. Для работы с командой нужно установить пакет gitk.

git log
Выводит историю фиксация для репозитория независимо от их публикации в восходящем репозитории.

git diff
Выводит построчные различия между локальным рабочим файлом и таким же файлом в понимании Git.

3.6. Лицензии

Поскольку проекты с открытым кодом публично доступны, они распространяются с теми или иными лицензиями. Эволюция лицензий для открытых (Open Source) и бесплатных программ (Free Software) представляет исторический интерес. Ниже приведены две ссылки на информацию о развитии и изменении таких лицензий.

- Open source license history.
- Free software license history.

В общем случае YP широко распространяется по лицензии Massachusetts Institute of Technology (MIT), разрешающей использовать программу в фирменных (proprietary) программах при включении в программу лицензии. Лицензия MIT совместима с GNU General Public License (GPL). Правки в YP обычно используют схему восходящего лицензирования. Информация о лицензии MIT доступна по ссылке, как и лицензия GNU GPL.

При создании образа с помощью YP процесс сборки использует известный список лицензий для обеспечения соответствия. Этот список можно найти в дереве источников (meta/files/common-licenses). По завершении сборки список всех найденных и использованных при сборке лицензий помещается в каталог сборки (tmp/deploy/licenses).

Если модуль требует лицензирования, не указанного в базовом списке, процесс сборки выдаёт предупреждение. Это упрощает разработчикам определение лицензий, с которыми следует распространять программы. Тем не менее, даже в этом случае окончательное решение о лицензировании остаётся за разработчиком.

Базовый список лицензий, используемый процессом сборки, является комбинацией списка SPDX[1] и проектов Open Source Initiative (OSI). SPDX Group является рабочей группой Linux Foundation, которая поддерживает спецификацию стандартного формата для передачи компонент, лицензий и авторских прав, связанных с программным пакетом. OSI является корпорацией, посвятившей себя определению исходного кода (OSD[2]) и усилиям по рассмотрению и утверждению лицензий, соответствующих OSD. Информацию, которая может помочь в поддержке соответствия лицензиям в жизненном цикле продукции, созданной с использованием YP, можно найти в разделе Maintaining Open Source License Compliance During Your Product's Lifecycle [1].

Глава 4. Концепции YP

В этой главе рассмотрены концепции, выходящие за рамки практических и справочных руководств. Описаны такие компоненты, как рабочий процесс системы сборки OE, инструменты кросс-разработки, кэш общих состояний и т. п.

4.1. Компоненты YP

Среда выполнения задач BitBake вместе с разными типами конфигурационных файлов образует OpenEmbedded-Core. В этом разделе рассматриваются указанные компоненты с описанием их использования и взаимодействий. BitBake обеспечивает синтаксический анализ и выполнение файлов данных, которые могут иметь несколько типов:

- *задания* обеспечивают детали для конкретных частей программ;
- *данные классов* содержат абстракции общей информации сборки (например, способ сборки ядра Linux);
- *данные конфигурации* задают машинозависимые настройки, правила и т. п., выступая в качестве склеивающей субстанции для сборки.

BitBake знает, как объединить множество источников данных и рассматривает каждый источник данных как уровень. Уровни описаны в разделе Understanding and Creating Layers [1]. Дополнительные сведения о взаимодействии базовых компонент приведены в разделе 4.3. Концепции системы сборки OE.

4.1.1. BitBake

BitBake является основным инструментом системы сборки OE и отвечает за синтаксический анализ метаданных, генерацию на их основе списка задач и последующее выполнение этих задач. В этом параграфе приведено краткое описание BitBake, а полную информацию можно найти в [7]. Для просмотра списка поддерживаемых BitBake можно воспользоваться командой bitbake -h или bitbake --help.

[1]Software Package Data Exchange - обмен данными о программных пакетах.
[2]Open Source Definition.

Чаще всего BitBake применяется в форме bitbake packagename, где packagename задаёт имя собираемого пакета (часто называется целью - target). Зачастую имя пакета совпадает с первой частью имени файла задания (например, foo для задания foo_1.3.0-r0.bb). Так, для обработки задания Smatchbox-desktop_1.2.3.bb следует ввести команду

```
$ bitbake matchbox-desktop
```

Может существовать несколько версий задания matchbox-desktop и BitBake выберет одно из них на основе конфигурации дистрибутива. Более подробно процедура выбора описана в разделе Preferences [7].

BitBake пытается заранее выполнить все задачи, от которых имеется зависимость. Например, перед сборкой matchbox-desktop программа BitBake может собирать кросс-компилятор и glibc, если они ещё не собраны.

Следует обратить внимание на опцию BitBake -k (или --continue), которая задаёт попытку продолжать работу даже после возникновения ошибки. Обычно ошибка прерывает сборку и задание, в котором она возникла, и зависящие от него задания не могут быть повторены, однако эта опция позволяет обработать другие зависимости.

4.1.2. Задания

Файлы с расширением .bb содержат задания (recipe). В общем случае задание включает данные об одной программе, включающие местоположение исходного кода, применяемые к нему исправления (если они имеются), особые опции настройки, способ компиляции исходных файлов и способ создания выходного пакета.

Иногда для обозначения заданий используется термин «пакет» (package), однако чаще пакетом называют сформированный результат системы сборки OE (т. е. файлы .ipk или .deb).

4.1.3. Классы

Файлы классов (.bbclass) содержат информацию, применимую к множеству заданий. Примером может служить класс autotools, который содержит общие настройки для программ с автоматической настройкой (Autotools). Работа с классами описана в разделе Classes [3].

4.1.4. Конфигурации

Конфигурационные файлы (.conf) определяет переменные, управляющие процессом сборки OE. Эти файлы размещаются в нескольких областях для настройки машины, дистрибутива, параметров компиляции, общих параметров конфигурации и пользовательских параметров в файле conf/local.conf каталога сборки.

4.2. Уровни

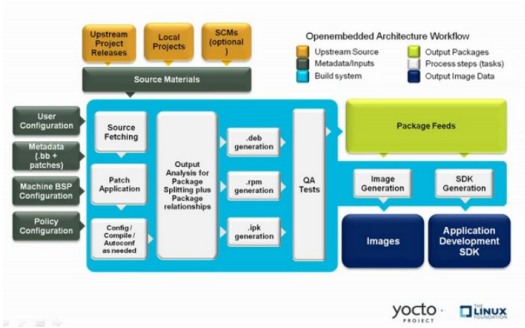
Уровнями называют репозитории связанных метаданных (наборов инструкций) указывающих системе сборки OE как собирать цель. Модель уровней YP облегчает совместную работу, обмен, настройку и многократное использование настроек в среде разработки YP. Уровни логически разделяют информацию разных проектов. Например, можно создать уровень для хранения всех конфигураций, связанных с определённой аппаратной платформой. Изоляцию аппаратно-зависимых конфигураций позволяет использовать другие метаданные совместно, используя свой уровень аппаратных метаданных для каждой целевой платформы.

В среде разработки YP имеется множество уровней, которые доступны по ссылкам Yocto Project Curated Layer Index и OpenEmbedded Layer Index.

По соглашению уровни в YP имеют определённую форму и соответствие заданной структуре позволяет BitBake принимать в процессе сборки допущения о местоположении разных типов метаданных. Процедуры и инструменты для работы с уровнями описаны в разделе Understanding and Creating Layers [1].

4.3. Концепции системы сборки OE

В этом разделе более подробно рассматриваются процессы, используемые системой сборки OE в среде YP. На рисунке представлен общий вид рабочего процесса сборки.



В общем случае рабочий процесс сборки включает несколько функциональных областей.

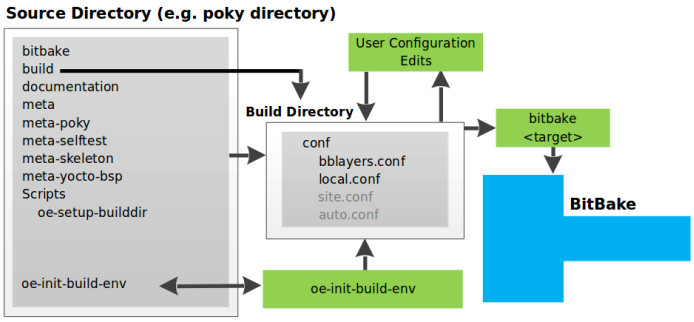
- *Пользовательская конфигурация* задаёт метаданные, используемые для управления процессом сборки.
- *Уровни метаданных* содержат информацию о программах, машине и дистрибутиве (метаданные).
- *Исходные файлы* с «восходящими» выпусками программ, локальными проектами и SCM.
- *Система сборки*, работающая под управлением BitBake. Этот блок определяет извлечение исходного кода, применение правок (patch), компиляцию, анализ вывода для создания пакетов, создание и проверку пакетов, генерацию образов, а также генерацию инструментов кросс-разработки.
- *Источники (хранилища) пакетов* - каталоги с выходными пакетами (RPM, DEB, IPK), которые используются при создании образа или SDK на выходе системы сборки. Эти хранилища могут копироваться и совместно

использоваться через web-сервер или иным путём для расширения или обновления имеющихся образов или устройств в процессе работы, если на них включено управление пакетами.

- *Образы*, создаваемые рабочим процессом.
- *SDK* - инструменты кросс-разработки, создаваемые BitBake вместе с образом или отдельно.

4.3.1. Пользовательская конфигурация

Пользовательская конфигурация помогает задать сборку, указывая BitBake целевую архитектуру, источники программного кода и другие параметры сборки, как показано на рисунке.



BitBake нужны некоторые базовые файлы конфигурации для выполнения сборки. Минимальный набор конфигурационных данных хранится в каталоге build/conf дерева источников, которое здесь для простоты называется каталогом Poky. При клонировании репозитория Poky или загрузке и распаковке выпуска YP можно указать для дерева источников любой желаемый каталог (здесь предполагается каталог poky).

Репозиторий Poky - это, прежде всего, объединение имеющихся репозиториев, а не канонический «восходящий» источник. Уровень meta-poky в Poky содержит каталог conf с примерами файлов конфигурации, которые могут служить основой для реальных файлов, применяемых сценарием oe-init-build-env для организации среды сборки. Выполнение этого сценария создает каталог сборки, если его ещё нет. BitBake использует каталог сборки при работе. Этот каталог включает подкаталог conf с принятыми по умолчанию файлами local.conf и bblayers.conf. Эти два файла создаются автоматически, если их ещё не было к моменту запуска сценария организации среды сборки.

Поскольку репозиторий Poky является объединением других репозиториев, некоторые пользователи могут применять сценарий инициализации (oe-init-build-env) в контексте репозиториев OpenEmbedded-Core и BitBake, а не Poky. В зависимости от использованного сценария вызываются разные субсценарии организации каталога сборки (Yocto или OE). В частности, сценарий scripts/oe-setup-builddir в каталоге poky организует каталог сборки и при необходимости помещает в него файлы, подходящие для среду разработки YP. Этот сценарий использует переменную $TEMPLATECONF для определения конфигурационных файлов, которые нужно найти.

Файл local.conf содержит множество переменных, задающих среду сборки, часть которых перечислена ниже. Конфигурационный файл local.conf, создаваемый по умолчанию сценарием настройки среды сборки, описан в файле local.conf.sample уровня meta-poky.

- MACHINE выбирает целевую машину.
- DL_DIR указывает каталог для загрузки файлов.
- SSTATE_DIR указывает каталог общих состояний.
- TMPDIR указывает каталог для вывода результатов сборки.
- DISTRO задаёт политику для дистрибутива.
- PACKAGE_CLASSES задаёт параметры создания пакетов.
- SDKMACHINE указывает целевую архитектуру для SDK.
- EXTRA_IMAGE_FEATURES задаёт дополнительные пакеты для включения в образ.

Эти параметры конфигурации можно задать также в файлах conf/site.conf и conf/auto.conf.

Файл bblayers.conf указывает системе сборки BitBake уровни, включаемые в процесс. По умолчанию этот файл содержит минимальный набор уровней, требуемых для сборки, однако в него можно вручную добавить любые нужные уровни. Работа с файлом bblayers.conf описана в разделе Enabling Your Layer [1].

Файлы site.conf и auto.conf сценарий организации среды не создает. Файл site.conf можно создать вручную, а файл auto.conf обычно создает autobuilder.

- Файл site.conf можно использовать для настройки нескольких каталогов сборки. Например, при наличии нескольких сред сборки с общими свойствами здесь можно указать используемые по умолчанию свойства сборки. Хорошим примером служит задание формата пакетов через переменную PACKAGE_CLASSES.

  Одним из вариантов применения файла conf/site.conf является преобразование переменной BBPATH для включения пути в conf/site.conf. Тогда BitBake при просмотре метаданных с использованием BBPATH найдёт файл conf/site.conf и применит заданную в нем конфигурацию. Для переопределения конфигурации в отдельном каталоге сборки нужные параметры задаются в файле conf/local.conf внутри этого каталога.

- Файл auto.conf обычно создает autobuilder, помещая в него настройки из conf/local.conf или conf/site.conf.

Конфигурационные файлы можно редактировать для уточнения или добавления новых параметров.

При запуске сборки командой bitbake target программа BitBake сортирует конфигурации и в конечном итоге задаёт среду сборки. Важно понимать, что система сборки OE читает файлы конфигурации в определённом порядке - site.conf,

auto.conf и local.conf. При этом применяются обычные правила для операторов присваивания, описанные в разделе Syntax and Operators [7]. Благодаря заданному порядку анализа файлов конфигурации, можно задавать разные значения переменных. Например, в файлах auto.conf и local.conf могут быть заданы разные значения переменной variable1, а поскольку файл local.conf анализируется после auto.conf, variable1 получит значение из файла local.conf.

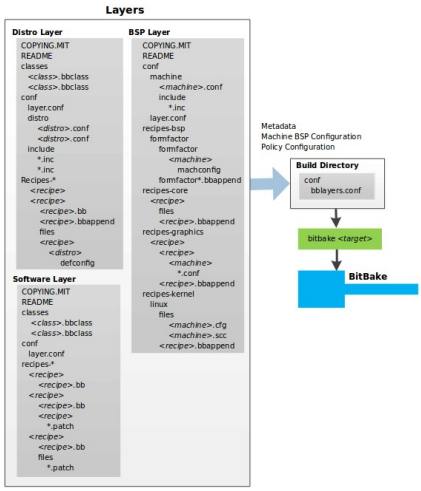
4.3.2. Метаданные, конфигурация машины и политики

Выше была описана пользовательская конфигурация, определяющая глобальное поведение BitBake. Здесь рассматриваются уровни, которые система сборки использует для дальнейшего управления процессом. Эти уровни включают метаданные для программ, машины и политики (правил).

В общем случае выделяют три три типа входной информации для уровней, описанных ниже.

- *Метаданные (*.bb + *правки).* Уровни программ включают указанные пользователем задания, файлы исправлений (patch) и добавлений. Хорошим примером может служить уровень meta-qt5 из OpenEmbedded Layer Index. Этот уровень создан для версии 5.0 популярной среды кроссплатформенной разработки Qt для настольных, встраиваемых и мобильных систем.
- *Конфигурация BSP для машины.* Уровни пакетов поддержи плат (BSP) задают машинозависимые конфигурации, относящиеся к конкретной целевой архитектуре. Хорошим примером служит уровень BSP из эталонного дистрибутива Poky (meta-yocto-bsp).
- *Конфигурация политики.* Уровни дистрибутивов (Distro) обеспечивают базовые правила для образа или SDK, собираемого с конкретным дистрибутивом. Например, в эталонном дистрибутиве Poky уровнем distro служит meta-poky. На уровне distro каталог conf/distro содержит конфигурационный файл дистрибутива (например, poky.conf).

Отмеченные выше уровни показаны на рисунке.



Обычно все уровни используют похожую структуру, включая файл лицензии (например, COPYING.MIT), если уровень является распространяемым, файл README (хороший тон особенно для распространяемых уровней), каталог конфигурации и каталоги заданий. Общая структура уровней YP описана в разделе Creating Your Own Layer [1]. Применение уровней кратко рассмотрено в разделах 4.2. Уровни и 2.2. Модель уровней YP, где отмечено, что в некоторых областях может существовать множество уровней, работающих в YP. В Source Repositories уровни выделены в категорию Yocto Metadata Layers. Следует отметить, что в YP Source Repositories имеются уровни, не представленные в OpenEmbedded Layer Index, - это устаревшие или экспериментальные уровни.

BitBake применяет файл conf/bblayers.conf, являющийся частью пользовательской конфигурации, для поиска уровней, включаемых в сборку.

4.3.2.1. Уровень дистрибутива

Уровень дистрибутива задаёт правила для дистрибутива. Хорошим тоном является отделение этой конфигурации от пользовательских уровней. Параметры в conf/distro/distro.conf переопределяют некоторые установки, которые BitBake берет из файла conf/local.conf в каталоге сборки. Ниже приведено краткое описание содержимого уровня дистрибутива.

- *Файлы классов* (.bbclass) задают общие функции, которые могут применяться заданиями дистрибутива. При наследовании класса заданием, оно получает все настройки и функции класса. Дополнительная информация о классах приведена в разделе Classes [3].
- *Параметры конфигурации* включают файлы конфигурации уровня (conf/layer.conf) и дистрибутива (conf/distro/*distro*.conf), а также включаемые для дистрибутива файлы.
- *Каталоги recipes-** содержат задания и дополнения, влияющие на функциональность дистрибутива. Здесь размещаются файлы заданий и дополнений со специфической для дистрибутива конфигурацией, сценариями инициализации, заданиями для пользовательских образов и т. п. Примерами каталогов recipes-* служат recipes-core и recipes-extra. Иерархия и содержимое каталогов recipes-* могут меняться. Обычно здесь содержатся файлы заданий (*.bb) и добавления (*.bbappend), каталоги с конфигурационными файлами конкретного дистрибутива и т. п.

4.3.2.2. Уровень BSP

Уровень BSP включает конфигурации машин для целевых аппаратных платформ, задавая все, что относится к машине, для которой собирается образ или SDK. Для уровня определена общая структура и форма [4]. Для того, чтобы уровень BSP был совместим с YP, нужно выполнить ряд структурных требований.

Каталог конфигурации уровня BSP содержит файлы для машины (conf/machine/machine.conf) и уровня (conf/layer.conf). Остальная часть уровня относится к заданиям, разделенным по функциям - recipes-bsp, recipes-core, recipes-graphics, recipes-kernel и т. п.. Могут присутствовать метаданные для разных форм-факторов, графических подсистем и т. п.

4.3.2.3. Уровень программ

Уровень программ содержит метаданные для дополнительных пакетов, используемых при сборке. Этот уровень не включает метаданных, относящихся к дистрибутиву или машине, помещённых в соответствующие уровни. Уровень включает все задания, файлы дополнений и правки (patch), которые нужны для проекта.

4.3.3. Источники

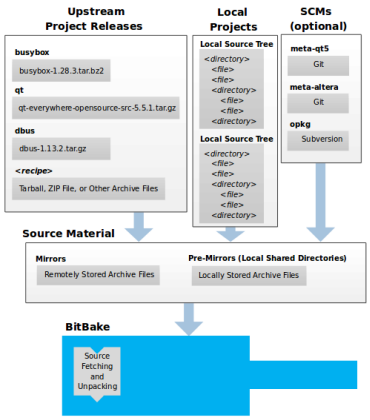
Чтобы система сборки OE могла создать образ или иную цель, она должна иметь доступ к исходным файлам. Метод организации исходных файлов зависит от проекта. Например, для выпущенных программ в проектах обычно используются архивы (tarball или иные файлы), которые фиксируют определённый выпуск. Для более динамичных и экспериментальных проектов могут применяться исходные файлы из репозитория SCM (например, Git). Извлечение файлов из репозитория позволяет контролировать состояние (выпуск) кода, который применяется для сборки. Может применяться и комбинация упомянутых вариантов, позволяющая потребителю выбрать источник файлов.

BitBake использует переменную SRC_URI для указания исходных файлов независимо от места их размещения. Эта переменная должна указываться в каждом задании.

Другим важным параметром выбора исходных файлов служит переменная DL_DIR, указывающая область кэширования с ранее загруженными файлами. Можно также задать системе сборки OE (переменная BB_GENERATE_MIRROR_TARBALLS) создание архивов (tarball) репозиториев Git (по умолчанию не выполняется) и запись их в DL_DIR.

Разумное использование каталога DL_DIR может сократить время сборки при использовании файлов из Internet. Разумно размещать каталог DL_DIR за пределами каталога сборки, это позволит при необходимости удалить каталог сборки, сохранив загруженные из сети исходные файлы.

Пример размещения исходных файлов представлен на рисунке.



4.3.3.1. Выпуски исходных проектов

Выпуски исходных проектов, хранящиеся где-либо в виде архивов (например, tarball или zip). Эти файлы соответствуют отдельным заданиям. Например, на рисунке выше представлены выпуски для BusyBox, Qt и Dbus. Архив может содержать любую программу, которая может быть собрана с использованием задания.

4.3.3.2. Локальные проекты

Локальные проекты предоставляются пользователем и размещаются локально, возможно в каталоге, где пользователь проверяет элементы (например, каталог, с деревом исходных файлов, используемым группой разработки). Каноническим методом включения локальных проектов является использование класса externalsrc. Можно использовать файл local.conf или файл дополнения задания для переопределения или установке в задании каталога с файлами локального проекта.

4.3.3.3. Работа с SCM

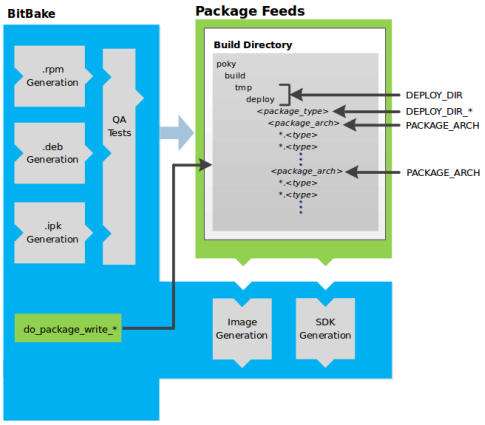
Ещё одним вариантом получения системой сборки исходных кодов является применение сборщиков (fetcher), работающих с системами SCM, такими как Git или Subversion, которые клонируют или выбирают (checked out) репозиторий. Задача do_fetch в BitBake использует переменную SRC_URI и префикс аргумента для определения конкретного сборщика. Система сборки OE может создавать архивы для репозиториев Git и помещать их в каталог DL_DIR, как описано в разделе BB_GENERATE_MIRROR_TARBALLS [3]. При извлечении из репозиториев BitBake применяет переменную SRCREV для определения конкретного выпуска.

4.3.3.4. Зеркала исходного кода

Поддерживаются два типа зеркал - предварительные и обычные, на которые указывают переменные PREMIRRORS и MIRRORS. соответственно. BitBake просматривает предварительные зеркала до поиска восходящих источников. Такие зеркала удобны при наличии общего каталога, который не указан в переменной DL_DIR (обычно это общий каталог внутри организации). Обычным зеркалом может быть любой сайт в Internet, служащий дополнительным местом размещения исходного кода и используемый при недоступности основного источника.

4.3.4. Хранилища пакетов

При генерации системой сборки образа или SDK, она берет пакеты из хранилища в каталоге сборки.



Хранилище содержит промежуточные результаты сборки. Система сборки OE включает классы для создания разных типов пакетов и нужно указать нужный класс в переменной PACKAGE_CLASSES. Перед размещением пакетов в хранилище система сборки проверяет качество полученного вывода с использованием класса insane.

Хранилище пакетов размещается в каталоге сборки, который задаётся комбинацией переменных и применяемым менеджером пакетов.

- DEPLOY_DIR задаётся как tmp/deploy в каталоге сборки.
- DEPLOY_DIR_* зависят от используемого менеджера пакетов. Для пакетов RPM, IPK, DEB или архивов используются переменные DEPLOY_DIR_RPM, DEPLOY_DIR_IPK, DEPLOY_DIR_DEB, DEPLOY_DIR_TAR.
- PACKAGE_ARCH определяет зависимые от архитектуры подкаталоги. Например, пакет может создаваться для архитектуры i586 или qemux86.

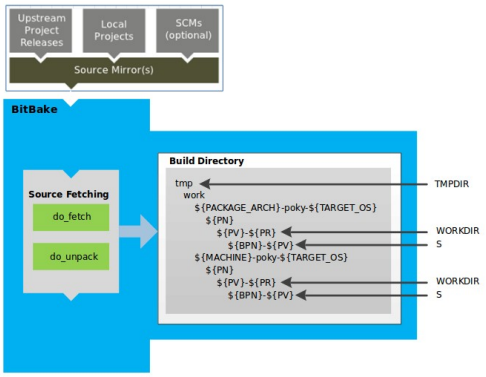
BitBake применяет задачи do_package_write_* для создания пакетов и размещения их в хранилище (например, do_package_write_ipk для IPK). Описания задач приведены в разделах do_package_write_deb, do_package_write_ipk, do_package_write_rpm и do_package_write_tar [3]. Например, при использовании менеджера IPK и создании пакетов для архитектуры i586 и qemux86 пакеты будут размещены в build/tmp/deploy/ipk/i586 и build/tmp/deploy/ipk/qemux86.

4.3.5. BitBake

Система сборки OE применяет BitBake для создания образов и SDK. Процесс работы BitBake включает несколько этапов, рассмотренных ниже. Документация BitBake представлена в [7].

4.3.5.1. Извлечение исходных кодов

Первым этапом сборки задания является извлечение и распаковка исходного кода, как показано на рисунке.



Задачи do_fetch и do_unpack извлекают исходные файлы и распаковывают их в каталоге сборки. Для каждого локального файла (например, file:///) из переменной SRC_URI в задании система сборки OE берет контрольную сумму файла для задания и помещает её в подпись для задачи do_fetch. Если локальный файл изменён, задача do_fetch и все зависящие от неё задачи запускаются заново.

По умолчанию все выполняется в каталоге сборки, имеющем определённую структуру, описанную в разделе build/ [3].

Каждое задание имеет область в каталоге сборки, где размещаются распакованные исходные файлы, указываемую переменной S. Имя каталога для любого конкретного задания определяется несколькими переменными.

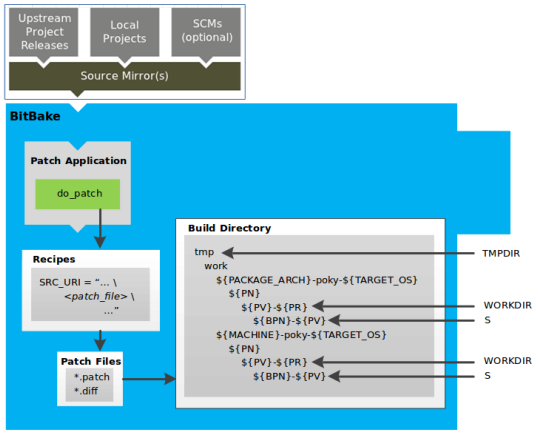
- TMPDIR - базовый каталог, где система OE выполняет всю сборку (по умолчанию каталог tmp).
- PACKAGE_ARCH - архитектура для сборки пакетов, которая может зависеть от конечной цели (например, архитектуры машины, хоста сборки, SDK или конкретной машины).
- TARGET_OS - операционная система целевого устройства (обычно linux, например qemux86-poky-linux").
- PN - имя задания для сборки пакета (в ином контексте смысл переменной может меняться).
- WORKDIR - место, где система OE собирает задание (т. е. выполняет работу по созданию пакета).
    - PV - версия задания, используемая для сборки пакета.
    - PR - выпуск (revision) задания, используемый для сборки пакета.
- S - каталог с распакованными исходными файлами для задания.

- BPN - имя задания для сборки пакета (вариант переменной PN без общих префиксов и суффиксов).
- PV - версия задания, используемая для сборки пакета.

Следует отметить на приведённом выше рисунке две иерархии - на основе архитектуры пакета (PACKAGE_ARCH) и машины (MACHINE). Нижележащие структуры идентичны. Различие будет определяться тем, что машина OE считает целью сборки (например, общая архитектура, хост сборки, SDK или конкретная машина).

4.3.5.2. Применение правок

После извлечения и распаковки исходного кода BitBake находит patch-файлы и применяет их к источникам.

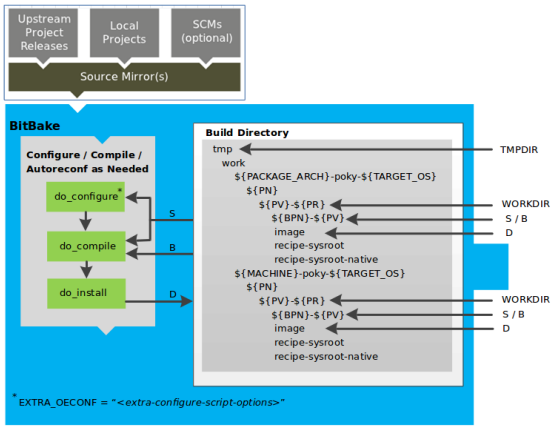


Задача do_patch использует операторы SRC_URI в задании и переменную FILESPATH для поиска patch-файлов. По умолчанию предполагается, что такие файлы имеют расширение .patch или .diff, но можно использовать параметры SRC_URI для указания системе сборки иных способов поиска patch-файлов. BitBake находит и применяет файлы правок для одного задания в порядке их указания. Переменная FILESPATH задаёт принятый по умолчанию набор каталогов для поиска patch-файлов. Найденные файлы применяются к исходным файлам из каталога S.

Размещение исходных файлов описано в параграфе 4.3.5.1. Извлечение исходных кодов, создание и применение patch-файлов описано в разделе Patching Code [1]. Дополнительная информация содержится в разделе Use devtool modify to Modify the Source of an Existing Component [2] и Using Traditional Kernel Development to Patch the Kernel [9].

4.3.5.3. Настройка, компиляции и подготовка пакетов

После исправления исходных файлов BitBake выполняет задачи настройки и компиляции, по завершении которых файлы копируются во временную область хранения для подготовки к созданию пакетов.

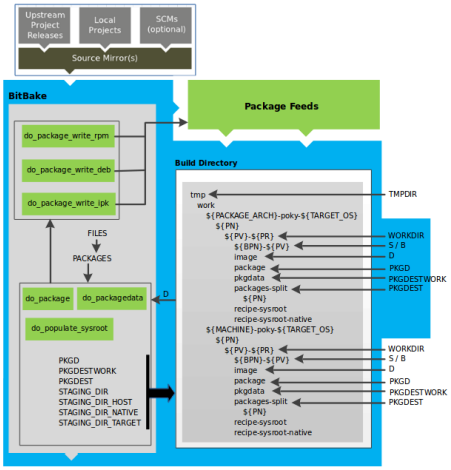


Эта часть процесса сборки включает несколько этапов.

- Задача do_prepare_recipe_sysroot организует до двух каталогов sysroot (recipe-sysroot и recipe-sysroot-native) в ${WORKDIR}, чтобы на этапе создания пакетов в sysroot могло быть содержимое задач do_populate_sysroot заданий, от которых зависит данное задание. Каталоги sysroot имеются для естественных (работают на хосте сборки) и целевых двоичных файлов.
- Задача do_configure настраивает конфигурацию исходного кода, включая и отключая соответствующие опции для собираемой программы. Настройка конфигурации может определяться самим заданием или унаследованным классом. Кроме того, конфигурацию может настраивать сама программа в зависимости от цели сборки. Конфигурации, обслуживаемые задачей do_configure относятся к исходному коду, собираемому заданием. При использовании класса autotools можно добавить опции с помощью переменных EXTRA_OECONF и PACKAGECONFIG_CONFARGS.
- Задача do_compile запускается после настройки конфигурации и выполняется в каталоге, заданном переменной B, который по умолчанию содержит каталог, указываемый переменной S.
- Задача do_install выполняется по завершении компиляции и копирует файлы из каталога B в область хранения, указанную переменной D, где позднее выполняется создание пакетов.

4.3.5.4. Разделение пакетов

После настройки, компиляции и подготовки кода система сборки анализирует результаты и делит вывод на пакеты.



Задачи do_package и do_packagedata совместно анализируют файлы в каталоге D и делят их на группы в соответствии с доступными пакетами и файлами. Анализ включает исключение символов отладки, поиск зависимостей от общих библиотек и связи между пакетами. Задача do_packagedata на основе анализа создает метаданные пакета, по которым система сборки генерирует пакеты. Задача do_populate_sysroot копирует нужную часть файлов, установленных задачей do_install, в нужный каталог sysroot. Для рабочих и промежуточных результатов анализа и разделения пакетов применяется несколько областей.

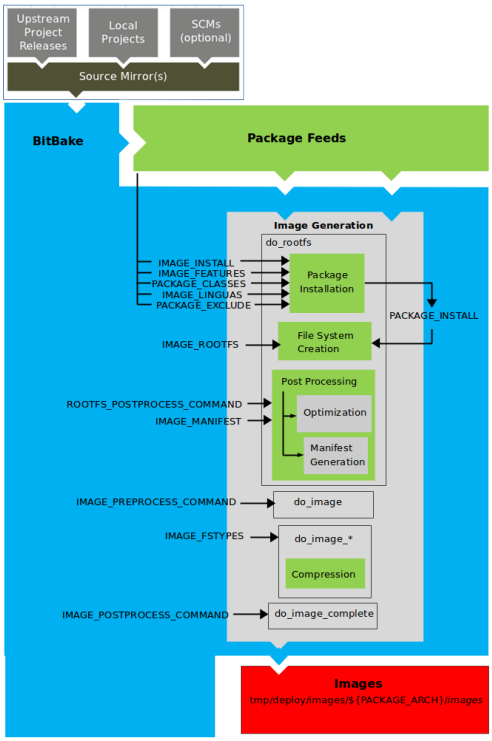
- PKGD - целевой каталог (package) для пакетов до их разделения.
- PKGDESTWORK - временная область (pkgdata), используемая задачей do_package для хранения метаданных.
- PKGDEST - родительский каталог (packages-split) для пакетов после разделения.
- PKGDATA_DIR - общий каталог глобального состояния, куда записываются метаданные, создаваемые процессом генерации пакетов. Процесс создания пакета копирует метаданные из PKGDESTWORK в каталог PKGDATA_DIR, где они становятся доступными глобально.
- STAGING_DIR_HOST - путь к sysroot для системы, на которой компонента будет работать (recipe-sysroot).
- STAGING_DIR_NATIVE - путь к sysroot с компонентами для сборочного хоста (recipe-sysroot-native).
- STAGING_DIR_TARGET - путь к sysroot с компонентами, собранными для выполнения в системе и генерирующими код для другой машины (например, задания cross-canadian).

Переменная FILES задаёт файлы, входящие в каждый пакет из PACKAGES.

В зависимости от типа создаваемых пакетов (RPM, DEB, IPK) задача do_package_write_* создает реальные пакеты и помещает их в хранилище ${TMPDIR}/deploy (см. параграф 4.3.4. Хранилища пакетов). Поддержки создания хранилищ непосредственно из каталогов deploy/* нет, поскольку для этого обычно требуется некий механизм загрузки новых пакетов в официальные хранилища (например, дистрибутив Ångström).

4.3.5.5. Создание образа

После разделения пакетов и записи их в хранилища (Package Feed) система сборки использует BitBake для создания образа файловой системы.



Процесс создания образа включает несколько этапов и зависит от ряда задач и переменных. Задача do_rootfs создает корневую файловую систему для образа, используя перечисленные ниже переменные.

- IMAGE_INSTALL - список базового набора пакетов для установки из хранилища (Package Feeds).
- PACKAGE_EXCLUDE - пакеты, которые не следует устанавливать в образ.
- IMAGE_FEATURES - свойства для включения в образ, большинство которых отображается в пакеты.
- PACKAGE_CLASSES указывает используемый менеджер пакетов (RPM, DEB, IPK), а затем помогает определить местонахождение пакетов в хранилище.
- IMAGE_LINGUAS определяет языки для которых устанавливаются дополнительные пакеты поддержки.
- PACKAGE_INSTALL задаёт окончательный список пакетов, передаваемый менеджеру для установки в образ.

Корневая файловая система создаётся с переменной IMAGE_ROOTFS, указывающей её местоположение, и переменной PACKAGE_INSTALL со списком пакетов для установки. инсталляция пакетов выполняется под управлением менеджера пакетов (dnf/rpm, opkg, apt/dpkg) независимо от того, включён ли менеджер для целевой платформы. В конце процесса, если менеджер пакетов не включён для целевой платформы, его файлы удаляются из корневой системы. На финальной стадии установки пакетов выполняются сценарии пост-установки. Все сценарии, которые не удалось выполнить на сборочном хосте, будут запущены при первой загрузке на целевой платформе. Если используется корневая файловая система с доступом только для чтения, все сценарии пост-установки должны быть выполнены на сборочном хосте в процессе инсталляции.

Пост-обработка выполняется на заключительных этапах задачи do_rootfs и включает оптимизацию и создание файла манифеста, который размещается в каталоге образа корневой файловой системы. В этом файле содержится построчный список установленных пакетов. Файл манифеста полезен, например, для класса testimage, чтобы решить вопрос о запуске соответствующих тестов. Процесс оптимизации запускается на созданном образе и включает mklibs, prelink и другие команды пост-обработки, в соответствии с переменной ROOTFS_POSTPROCESS_COMMAND. Процесс mklibs оптимизирует размер библиотек, а prelink - динамические ссылки общих библиотек для оптимизации времени загрузки исполняемых файлов.

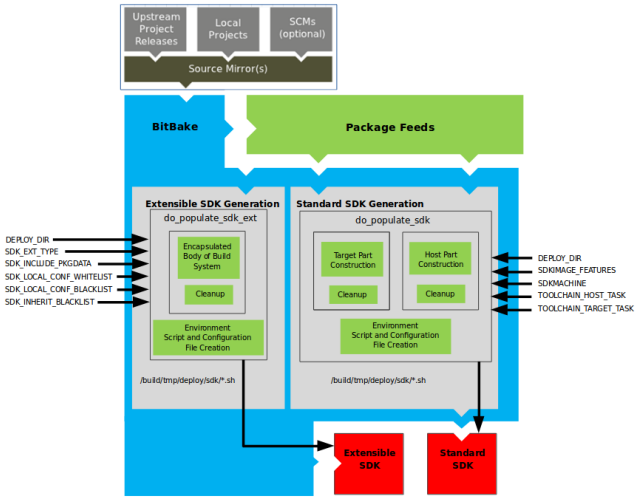
После создания корневой файловой системы начинается обработка образа задачей do_image. Система сборки запускает все команды предварительной обработки из переменной IMAGE_PREPROCESS_COMMAND, задающей список функций, вызываемых системой сборки перед созданием окончательных файлов образа.

Система сборки при необходимости динамически создает задачи do_image_* в соответствии с типами образов в переменной IMAGE_FSTYPES. Процесс превращает все в файл образа или устанавливает файлы образа и может сжимать образ корневой файловой системы. Форматы корневой файловой системы зависит от переменной IMAGE_FSTYPES, а сжатие зависит от его поддержки файловой системой. Например, динамически создаваемая задача при построении определённого типа образа может иметь вид do_image_type. Если в IMAGE_FSTYPES указан тип ext4, динамически создаваемая задача будет иметь форму do_image_ext4.

Финалом создания образа является задача do_image_complete, которая выполняет пост-обработку образа в соответствии с переменной IMAGE_POSTPROCESS_COMMAND, которая содержит список функций, вызываемых однократно при создании системой сборки окончательных файлов образа. Весь процесс генерации образа выполняется в рамках Pseudo, что позволяет задать корректных владельцев файлов корневой файловой системы.

4.3.5.6. Создание SDK

Система сборки OE использует BitBake для генерации сценариев установки SDK как для стандартных, так и для расширяемых SDK (eSDK).



Создание SDK описано в разделе 4.4. Создание инструментов кросс-разработки, а преимущества сборки с помощью задачи do_populate_sdk рассмотрены в разделе Building an SDK Installer [2].

Подобно созданию образа, сценарий установки SDK состоит из нескольких этапов и зависит от множества переменных. Задачи do_populate_sdk и do_populate_sdk_ext используют эти переменные для оказания помощи при создании списка реально устанавливаемых пакетов (см. 4.3.7. SDK для разработки приложений). Задача do_populate_sdk помогает создать стандартный SDK и обрабатывает целевую и хостовую часть. Целевая часть включает библиотеки и заголовочные файлы, а хостовая - SDK, работающий на SDKMACHINE. Задача do_populate_sdk_ext помогает собрать eSDK и обрабатывает хостовую и целевую части иначе, чем для стандартных SDK, инкапсулируя систему сборки, включающую все, что требуется для SDK (на хосте и целевой платформе). Независимо от типа SDK, задачи выполняют некоторую очистку, после чего создаётся сценарий организации среды кросс-разработки и все требуемые файлы конфигурации. Финальным выводом является сценарий установки инструментов кросс-разработки (.sh), включающий сценарий организации среды.

4.3.5.7. Файлы штампов и повторный запуск задач

Для каждой выполненной задачи BitBake записывает файл штампа в каталог STAMPS_DIR. Начало имени файла задаёт переменная STAMP, а остальная часть включает имя задачи и текущую входную контрольную сумму (4.5.2. Контрольные суммы (подписи)). Эта схема именования предполагает BB_SIGNATURE_HANDLER = "OEBasicHash", что выполняется почти всегда в текущей версии OE.

Для решения вопроса о повторном запуске задачи BitBake проверяет наличие для задачи штампа с совпадающей контрольной суммой. При наличии такого файла предполагается, что вывод задачи имеется и пригоден. Если такого файла нет, задача запускается снова. Механизм штампов является более общим, чем механизм кэширования общего состояния (sstate), описанный в параграфе 4.3.5.8. Пропуск задач и общее состояние. BitBake избегает повтора задач с действительным штампом в дополнение к задачам, которые можно ускорить с помощью кэша sstate. Однако следует понимать, что файлы штампов служат лишь маркерами проделанной работы и не содержат вывода задач. Реальный вывод размещается где-либо в TMPDIR (например, в WORKDIR задания). Механизм кэширования sstate добавляет способ кэширования вывода задач, который может совместно использоваться разными машинами сборки. Поскольку STAMPS_DIR обычно размещается в каталоге TMPDIR, удаление этого каталога приведёт к удалению STAMPS_DIR и задачи будут запускаться повторно для нового заполнения TMPDIR.

Если нужно признать ту или иную задачу устаревшей, можно пометить её флагом переменной nostamp. Если какая-либо задача зависит от такой задачи, она тоже будет считаться устаревшей (возможно, это не то, что вы хотите). Дополнительные сведения о подписях задач даны в разделе Viewing Task Variable Dependencies [1].

4.3.5.8. Пропуск задач и общее состояние

Приведённые выше описания задач предполагали, что BitBake нужно собирать все и нет созданного ранее вывода. В реальности BitBake позволяет пропустить задачи, если имеются созданные ранее объекты, которые обычно доступны в форме кэша общего состояния (sstate).

Идея задачи do_taskname_setscene заключается в том, что можно пропустить ту или иную задачу и просто поместить в нужное место соответствующие файлы, которые уже созданы. В некоторых случаях (например, для задач do_package_write_*) это имеет смысл, но в других задачах (например, do_patch или do_unpack) бесполезно, поскольку объем выполняемой работы не снижается.

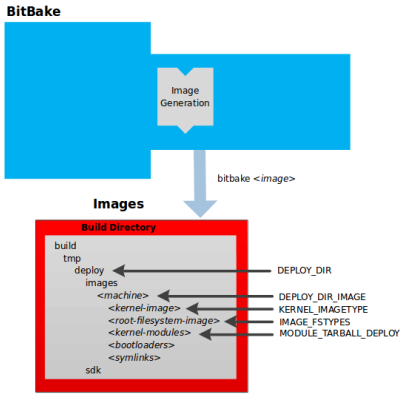
В системе сборки варианты пропуска (setscene) доступны чаще всего для задач do_package, do_package_write_*, do_deploy, do_packagedata и do_populate_sysroot, которые создают основную часть вывода системы сборки. Зависимости между задачами и их предшественниками известны системе сборки. Например, если BitBake запускает do_populate_sysroot_setscene для чего-либо, не имеет смысла выполнять любую из задач do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install. Однако при необходимости выполнения do_package BitBake придётся запускать и другие задачи. Ситуация усложняется при использовании кэша sstate, поскольку некоторые объекты просто не нужны. Например, может не требоваться компилятор или естественные инструменты, такие как quilt, если нечего компилировать или исправлять (patch). Если пакеты do_package_write_* доступны в sstate, BitBake не нужны данные задачи do_package.

С учётом приведённых сложностей BitBake работает в двух фазах. Сначала выполняется этап подготовки, где BitBake проверяет кэш sstate для всех целей, которые планируется собрать. BitBake выполняет быструю проверку наличия объектов без их загрузки. Если ничего не найдено, выполняется обычный процесс сборки. При наличии объекта в sstate система сборки работает в обратном направлении, от конечных целей, заданных пользователем. Например, при сборке образа система сначала ищет пакеты, нужные для создания образа. Если эти пакеты доступны, компилятор не требуется и даже не будет загружаться. Если что-то оказалось недоступным или возник отказ при загрузке или в задаче setscene, система сборки пытается установить зависимости (такие как компилятор) из кэша.

Доступность объектов кэша sstate определяется функцией, которая указана переменной BB_HASHCHECK_FUNCTION и возвращает список доступных объектов. Функция, указанная в BB_SETSCENE_DEPVALID, определяет необходимость выполнения данной зависимости.

4.3.6. Образы

Создаваемые системой сборки образы представляют собой сжатую форму корневой файловой системы, пригодной к загрузке на целевой платформе. Процесс создания образа показан на рисунке. Примеры образов, предоставляемых YP приведены в разделе Images [3].



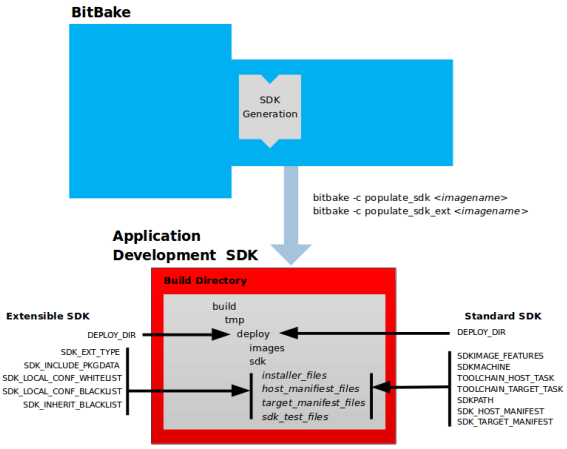
Образы записываются в каталог tmp/deploy/images/machine/ внутри каталога сборки. Здесь размещаются все файлы, предназначенные для загрузки на целевом устройстве. Переменная DEPLOY_DIR указывает каталог развёртывания, а DEPLOY_DIR_IMAGE - каталог с образами для текущей конфигурации.

- *Kernel-image* - двоичный файл ядра. Переменная KERNEL_IMAGETYPE определяет схему именования файлов с образами ядра. В каталоге deploy/images/machine может храниться множество образов ядра для машины.

- *Root-filesystem-image* - корневая файловая система для целевого устройства (например, файлы *.ext3 или *.bz2). Тип файловой системы задаёт переменная IMAGE_FSTYPES. В каталоге deploy/images/machine может храниться множество образов корневой файловой системы для машины.
- *Kernel-modules* - архив модулей для ядра, создание которого можно отключить установкой MODULE_TARBALL_DEPLOY = "0". В каталоге deploy/images/machine может храниться множество архивов модулей ядра для машины.
- *bootloaders* - загрузчики, поддерживающие образ, если целевая платформа позволяет это. В каталоге deploy/images/machine может храниться множество загрузчиков для машины.
- *symlinks* - каталог deploy/images/machine содержит символьные ссылки, указывающие наиболее свежие файлы для каждой машины. Ссылки могут быть полезны для внешних сценариев.

4.3.7. SDK для разработки приложений

Процессы создания SDK различаются для стандартного (например, bitbake -c populate_sdk imagename) и расширяемого SDK (например, bitbake -c populate_sdk_ext imagename).



Вывод представляет собой набор файлов, включающий самораспаковывающийся установщик SDK (*.sh), манифесты для хоста и цели, а также файлы для тестирования SDK. Пакет SDK включает инструменты кросс-разработки, набор библиотек и заголовочных файлов, а также сценарий организации среды SDK. Инструменты можно считать хостовой частью SDK, поскольку они применяются на машине разработки. Библиотеки и заголовки можно считать целевой частью, поскольку они собираются для целевой машины. Сценарий инициализации среды позволяет организовать рабочее окружение SDK.

YP поддерживает несколько методов организации среды кросс-разработки, включая загрузку собранного установщика SDK и сборку и инсталляцию своего SDK. Базовые сведения о кросс-разработке в среде YP приведены в разделе 4.4. Создание инструментов кросс-разработки, а организация среды рассмотрена в [2].

Выходные файлы для SDK сохраняются в каталоге deploy/sdk внутри каталога сборки, как показано на рисунке выше. Для расширяемого SDK перечисленные ниже переменные помогают настроить процесс.

- DEPLOY_DIR указывает каталог развёртывания (deploy).
- SDK_EXT_TYPE управляет копированием элементов общего состояния в расширяемый SDK (по умолчанию копируются).
- SDK_INCLUDE_PKGDATA управляет включением packagedata в расширяемый SDK для заданий цели world.
- SDK_INCLUDE_TOOLCHAIN управляет включением инструментария в расширяемый SDK.
- SDK_LOCAL_CONF_WHITELIST - список переменных для переноса из среды сборки в расширяемый SDK.
- SDK_LOCAL_CONF_BLACKLIST - список переменных, не переносимых из среды сборки в расширяемый SDK.
- SDK_INHERIT_BLACKLIST - список классов для глобального удаления из INHERIT в расширяемом SDK.

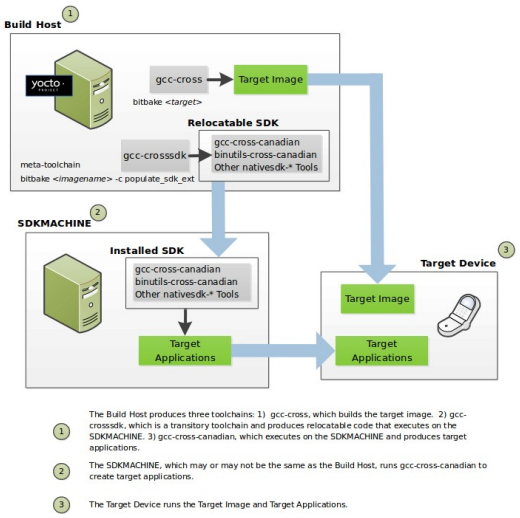
Приведённые ниже переменные связаны со стандартным SDK.

- DEPLOY_DIR указывает каталог развёртывания (deploy).
- SDKMACHINE - архитектура машины, где будут работать инструменты кросс-разработки.
- SDKIMAGE_FEATURES - список свойств для включения в целевую (target) часть SDK.
- TOOLCHAIN_HOST_TASK - список пакетов для хостовой части SDK (работают на SDKMACHINE). При использовании bitbake -c populate_sdk imagename для создания SDK применяется заданный по умолчанию набор, который можно изменить в этой переменной.
- TOOLCHAIN_TARGET_TASK - список пакетов для целевой части SDK (работают на целевой платформе).
- SDKPATH задаёт путь установки SDK, предлагаемый сценарием инсталляции.
- SDK_HOST_MANIFEST - список установленных пакетов хостовой части SDK.
- SDK_TARGET_MANIFEST - список установленных пакетов целевой части SDK.

4.4. Создание инструментов кросс-разработки

YP самостоятельно выполняет большую часть работы по созданию инструментов кросс-разработки. В этом разделе кратко описано создание и использование инструментов, а более полные сведения содержатся в [2].

В среде YP инструменты кросс-разработки служат для сборки образов и приложений, предназначенных для работы на целевой платформе. Система сборки OE создает нужные инструменты с помощью нескольких команд. На рисунке показан общий вид среды сборки и использования инструментария.



Большая часть работы выполняется на хосте сборки, используемом для создания образов и обычно работающем в среде YP. При запуске BitBake для создания образа система сборки OE использует компилятор gcc на хосте для создания кросс-компилятора gcc-cross, который применяется BitBake для компиляции исходных файлов при создании целевого образа. Можно считать gcc-cross автоматически генерируемым кросс-компилятором, который применяется только в BitBake. Расширяемый SDK не применяет gcc-cross-canadian, поскольку этот SDK содержит копию среды сборки OE и файловую систему sysroot с gcc-cross.

Цепочка событий при создании gcc-cross имеет вид

`gcc -> binutils-cross -> gcc-cross-initial -> linux-libc-headers -> glibc-initial -> glibc -> gcc-cross -> gcc-runtime`

- *gcc* - компилятор GNU Compiler Collection (GCC) на сборочном хосте.
- *binutils-cross* - минимальный набор двоичных утилит, требуемых для запуска фазы gcc-cross-initial.
- *gcc-cross-initial* - ранняя стадия процесса создания кросс-компилятора. На этом этапе создаётся gcc-cross, библиотека C и другие компоненты, требуемые для сборки на последующих этапах финального кросс-компилятора. Этот инструмент является естественным (native) пакетом (для хоста сборки).
- *linux-libc-headers* - заголовочные файлы, нужные кросс-компилятору.
- *glibc-initial* - изначальная версия библиотеки Embedded GNU C (GLIBC), нужная для создания glibc.
- *glibc* - библиотека GNU C.
- *gcc-cross* - финальная стадия процесса создания кросс-компилятора, который BitBake использует при сборке образа для целевого устройства.

  При замене инструментов кросс-компиляции потребуется заменить и gcc-cross. Этот инструмент также является естественным пакетом (работает на хосте сборки).
- *gcc-runtime* - используемые при работе (runtime) библиотеки из процесса создания инструментов. Этот инструмент создает двоичный файл с runtime-библиотеками для целевого устройства.

Можно с помощью системы сборки OE создать установщик переносимого SDK для разработки приложений. При запуске установщика будет инсталлироваться инструментарий (например, gcc-cross-canadian, binutils-cross-canadian, и другие инструменты nativesdk-*), который является естественным для SDK (т. е. для SDK_ARCH), и нужно собирать и тестировать программы. На рисунке выше показаны команды, позволяющие собрать эти инструменты. Эти средства кросс-разработки собираются для работы на SDKMACHINE, которая может, но не обязана быть хостом сборки.

Если целевая архитектура поддерживается YP, можно воспользоваться готовыми образами из комплекта YP, где уже включены установщики инструментов кросс-разработки.

Процесс создания переносимого инструментария имеет вид

`gcc -> binutils-crosssdk -> gcc-crosssdk-initial -> linux-libc-headers -> glibc-initial -> nativesdk-glibc -> gcc-crosssdk -> gcc-cross-canadian`

- *gcc* - компилятор GNU Compiler Collection (GCC) на сборочном хосте.
- *binutils-crosssdk* - минимальный набор двоичных утилит, требуемых для запуска фазы gcc-crosssdk-initial.
- *gcc-cross-initial* - ранняя стадия процесса создания кросс-компилятора. На этом этапе создаётся gcc-cross, библиотека C и другие компоненты, требуемые для сборки на последующих этапах финального кросс-компилятора. Этот инструмент является естественным (native) пакетом (для хоста сборки).
- *linux-libc-headers* - заголовочные файлы, нужные кросс-компилятору.
- *glibc-initial* - изначальная версия библиотеки Embedded GNU C (GLIBC), нужная для создания nativesdk-glibc.

- *nativesdk-glibc* — библиотека Embedded GLIBC, нужная для создания gcc-crosssdk.
- *gcc-crosssdk* - финальный этап процесса создания переносимого кросс-компилятора, который является временным и не покидает сборочный хост. /тот компилятор помогает создать компилятор gcc-cross-canadian compiler, который можно переносить. Этот инструмент является естественным пакетом (на хосте сборки).
- *gcc-cross-canadian* - окончательный переносимый кросс-компилятор. При работе на SDKMACHINE этот инструмент создает исполняемый код для целевого устройства. Создаётся лишь один компилятор cross-canadian для архитектуры, поскольку эти компиляторы могут выполнять оптимизацию для разных процессоров с использованием конфигурации, передаваемой компилятору в командах. Это позволяет обойтись одним компилятором и снижает размер инструментария.

Информация о сборке установщика инструментов кросс-разработки приведена в разделе Building an SDK Installer [2].

4.5. Общий кэш состояний

Система сборки OE создает все с нуля, если только BitBake не указывает, что некоторые части не нужно собирать заново. Создание с нуля привлекательно, поскольку все собранные компоненты будут свежими и не останется устаревших данных, которые могут вызвать проблемы. Обычно разработчики при обнаружении проблем начинают сборку с «чистого листа», чтобы знать исходное состояние. Однако сборка с нуля кроме преимуществ имеет и недостатки, связанные с существенным расходом времени на сборку.

В YP реализован код общего состояния, поддерживающий инкрементную сборку. Такая сборка зависит от ответов на приведённые ниже вопросы.

- Какая часть системы была изменена, а что сохранилось неизменным?
- Были изменённые части удалены или заменены?
- Как будут использоваться собранные ранее неизменные компоненты, если они доступны?

По первому вопросу система сборки находит изменения на «входах» в данную задачу через контрольные суммы (подписи) ввода. Если подпись изменилась, система считает входные данные изменёнными с необходимостью повтора задачи. По второму вопросу код общего состояния (sstate) отслеживает вывод задач в процессе сборки, который может быть удалён или обновлён. Третий вопрос частично решается вместе со вторым в предположении, что система сборки может извлекать объекты sstate из удалённого хранилища и устанавливать пригодные объекты.

- Система сборки не поддерживает информацию PR как часть общего состояния пакетов, поэтому существуют соображения, влияющие на поддержку подачи общего состояния. Информация о работе системы сборки с пакетами и возможности отслеживания роста PR приведена в разделе Automatically Incrementing a Binary Package Revision Number [1].
- Код поддержки инкрементной сборки достаточно сложен. Методы, помогающие обойти проблемы, связанные с кодом общего состояния, рассмотрены в разделах Viewing Metadata Used to Create the Input Signature of a Shared State Task и Invalidating Shared State to Force a Task to Run [1].

4.5.1. Общая архитектура

При определении частей, требующих повторной сборки, BitBake работает на уровне задач, а не заданий. Для разъяснения такого подхода рассмотрим ситуацию, когда был включён менеджер пакетов IPK, а затем произошёл переход на DEB. В этом случае вывод задач do_install и do_package остаётся пригодным, однако при работе на уровне заданий сборка не будет включать файлов .deb и придётся повторить её целиком, что не является эффективным решением.

4.5.2. Контрольные суммы (подписи)

Код общего состояния использует контрольную сумму, которая служить уникальной подписью ввода в задачу, для определения необходимости повторного запуска. Поскольку изменение ввода вызывает перезапуск, процесс должен обнаруживать изменение любого ввода в данную задачу. Для задач оболочки это просто, поскольку процесс сборки генерирует сценарий run для каждой задачи и можно создать контрольную сумму, позволяющую увидеть изменения.

Однако в реальности задача сложнее, поскольку в контрольную сумму следует включать не все. Например, реальный путь сборки WORKDIR не имеет значения и изменение каталога не должно влиять на вывод для целевых пакетов. Кроме того, процесс сборки может создавать переносимые естественные и кросс-пакеты. Оба варианта пакетов запускаются на хосте сборки, однако вывод кросс-пакетов предназначен для целевой архитектуры. Поэтому из контрольной суммы следует исключить WORKDIR. Проще всего это сделать, задав для WORKDIR некое фиксированное значение, и создавать контрольную сумму для сценария run.

Другая проблема заключается в том, что сценарий run содержит функции, которые могут не вызываться. Решение для инкрементной сборки включает код, вычисляющий зависимости между shell-функциями. Этот код служит для сокращения сценариев run до минимального набора, смягчая проблему и делая сценарии run более читаемыми.

Для сценариев Python применяется такой же подход даже в более сложных задачах. Процесс должен выяснить, к каким переменным обращается функция Python и зависимости между функциями, а затем создавать контрольную сумму для входных данных задачи. Подобно WORKDIR, существуют ситуации, где зависимости следует игнорировать. Это можно задать с помощью строки вида PACKAGE_ARCHS[vardepsexclude] = "MACHINE". Здесь предполагается, что переменная PACKAGE_ARCHS не зависит от значения MACHINE даже при наличии ссылки на неё. Точно также имеются случаи, когда нужно добавить зависимости, которые BitBake не может найти. Это можно сделать в форме строки PACKAGE_ARCHS[vardeps] = "MACHINE", которая явно указывает зависимость PACKAGE_ARCHS от MACHINE.

Возможен случай с Python, где BitBake не может определить зависимости. При работе в режиме отладки (-DDD), BitBake выдаёт сообщения при обнаружении невозможности выяснить зависимости. Команда YP пока не нашла решения этой проблемы, но занимается его поиском.

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

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

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

В примере не указана переменная WORKDIR, поскольку она задаёт путь внутри TMPDIR, уже имеющейся в списке.

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

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

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

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

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

4.5.3. Общее состояние

Контрольные суммы и зависимости решают проблему поддержки общего состояния наполовину. Другая половина связана с возможностью использовать данные контрольной суммы в процессе сборки для повторного использования или повторной сборки конкретных компонент. Класс sstate является сравнительно общей реализацией создания «моментального снимка» данной задачи. Идея состоит в том, чтобы процесс сборки мог не думать об источнике вывода задачи. Это могут быть недавно собранные файлы или файлы, загруженные извне и распакованные. Имеется два типа вывода, один из которых является созданием каталога в WORKDIR. Примером может служить вывод задачи do_install или do_package. Другой тип связан с объединением вывода в общем дереве каталогов, таком как sysroot.

Команда YP пыталась сохранить детали реализации спрятанными в классе sstate. С точки зрения пользователя добавление переноса спрятанного общего состояния в задачу так же просто, как в примере do_deploy из класса.

```
DEPLOYDIR = "${WORKDIR}/deploy-${PN}"
SSTATETASKS += "do_deploy"
do_deploy[sstate-inputdirs] = "${DEPLOYDIR}"
do_deploy[sstate-outputdirs] = "${DEPLOY_DIR_IMAGE}"

python do_deploy_setscene () {
    sstate_setscene(d)
}
addtask do_deploy_setscene
do_deploy[dirs] = "${DEPLOYDIR} ${B}"
do_deploy[stamp-extra-info] = "${MACHINE_ARCH}"
```

Ниже приведены разъяснения к примеру.

- Добавление do_deploy к SSTATETASKS вносит требование дополнительной обработки, связанной с sstate, которая реализуется в классе sstate до задачи do_deploy или после неё.
- Назначение do_deploy[sstate-inputdirs] = "${DEPLOYDIR}" указывает, что do_deploy при обычной работе (без кэша sstate) помещает свой вывод в ${DEPLOYDIR} и он становится вводом кэша общего состояния.
- Строка do_deploy[sstate-outputdirs] = "${DEPLOY_DIR_IMAGE}" ведёт к копирования содержимого кэша общего состояния в ${DEPLOY_DIR_IMAGE}.

  Если do_deploy ещё нет в кэше общего состояния или входная контрольная сумма (подпись) изменилась с момента кэширования вывода, запускается задача для заполнения кэша общего состояния, после чего содержимое кэша копируется в ${DEPLOY_DIR_IMAGE}. Если do_deploy есть в кэше и контрольная сумма действительна (вводы имеющих отношение к делу задач не изменились), содержимое кэша напрямую копируется в ${DEPLOY_DIR_IMAGE} задачей do_deploy_setscene task с пропуском задачи do_deploy.
- Приведённая ниже задача обеспечивает логику склейки для предыдущих установок.

```
python do_deploy_setscene () {
    sstate_setscene(d)
```

```
}
addtask do_deploy_setscene
```

Задача sstate_setscene() принимает указанные выше флаги как входные данные и ускоряет do_deploy за счёт кэша состояний, если это возможно. Если ускорение удалось, sstate_setscene() возвращает True. В противном случае возвращается False и выполняется обычная задача do_deploy. Дополнительная информация приведена в разделе setscene [7].

- Строка do_deploy[dirs] = "${DEPLOYDIR} ${B}" создает ${DEPLOYDIR} и ${B} до запуска задачи do_deploy, а также устанавливает ${B} в качестве рабочего каталога do_deploy. Дополнительная информация приведена в разделе Variable Flags [7].

  Если sstate-inputdirs и sstate-outputdirs совпадают, можно использовать sstate-plaindirs. Например, для сохранения вывода ${PKGD} и ${PKGDEST} из задачи do_package можно использовать

  ```
  do_package[sstate-plaindirs] = "${PKGD} ${PKGDEST}"
  ```

- Строка do_deploy[stamp-extra-info] = "${MACHINE_ARCH}" добавляет метаданные в файл штампа. В этом случае метаданные делают задачу специфичной для архитектуры машины. Описание флага stamp-extra-info приведено в разделе [7].

- Переменные sstate-inputdirs и sstate-outputdirs могут включать несколько каталогов. Например, переменные PKGDESTWORK и SHLIBWORK объявлены ниже как входные для общего состояния, а PKGDATA_DIR и SHLIBSDIR - как соответствующие выходные каталоги.

  ```
  do_package[sstate-inputdirs] = "${PKGDESTWORK} ${SHLIBSWORKDIR}"
  do_package[sstate-outputdirs] = "${PKGDATA_DIR} ${SHLIBSDIR}"
  ```

- Эти методы также позволяют взять файл блокировки при работе со структурами кэша общих состояний для случаев, когда важно добавление или удаление файлов

  ```
  do_package[sstate-lockfile] = "${PACKAGELOCK}"
  ```

Код общего состояния просматривает файлы состояний по переменным SSTATE_DIR и SSTATE_MIRRORS.

```
SSTATE_MIRRORS ?= "\
file://.* http://someserver.tld/share/sstate/PATH;downloadfilename=PATH \n \
file://.* file:///some/local/dir/sstate/PATH"
```

Каталог общего состояния (SSTATE_DIR) содержит подкаталоги, имена которых образованы двумя первыми символами хэш-значения. Если структура каталога общего состояния для зеркала совпадает со структурой SSTATE_DIR, нужно задать "PATH" как часть URI, чтобы разрешить системе сборки сопоставление с подходящим каталогом.

Достоверность общего состояния можно проверить просто по имени файла, поскольку оно содержит контрольную сумму (подпись), как указано выше. Если найдено действительное общее состояние, процесс сборки загружает его и применяет для ускорения задачи.

Процессы сборки применяют задачи *_setscene в фазе ускорения задач, которую BitBake проходит перед основным кодом, пытаясь ускорить все задачи, для которых можно найти общее состояние. Если такое состояние найдено, это позволяет пропустить задачу вместе с задачами, от которых она зависит.

4.6. Автоматически добавляемые зависимости при работе

Система сборки OE автоматически добавляет общие типы зависимостей между пакетами при работе, избавляя от необходимости включать их в RDEPENDS. Имеется три автоматических механизма (shlibdeps, pcdeps, depchains), которые обслуживают общие библиотеки, конфигурацию пакетов (pkg-config), а также пакеты разработки и отладки (-dev и -dbg). Остальные типы зависимостей должны указываться вручную.

- *shlibdeps.* При выполнении задачи do_package для каждого задания отмечаются все общие библиотеки, устанавливаемые заданием. Для каждой такой библиотеки содержащий её пакет регистрируется как поставщик библиотеки (soname). Полученное сопоставление между пакетами и общими библиотеками (shared-library-to-package) сохраняется глобально в переменной PKGDATA_DIR задачей do_packagedata.

  Одновременно все исполняемые файлы и библиотеки, устанавливаемые заданием, проверяются на предмет связи с общими библиотеками. Для каждой найденной зависимости проверяется PKGDATA_DIR для обнаружения присутствия пакетов (вероятно из других заданий), предоставляющих библиотеку. Если такой пакет найден, добавляется зависимость при работе от содержащего библиотеку пакета.

  Автоматически добавляемые зависимости учитывают также ограничения по версиям. Это ограничение указывает, что должна применяться версия не ниже текущей версии пакета, предоставляющего библиотеку, как будто добавлено условие "package (>= version)" в RDEPENDS. Это заставляет при необходимости обновлять пакет, содержащий библиотеку.

  Если нужно исключить регистрацию пакета в качестве поставщика общей библиотеки (например, библиотека предназначена лишь для внутреннего применения), следует добавить PRIVATE_LIBS в задание для пакета.

- *pcdeps.* При выполнении задачи do_package для каждого задания фиксируются все модели pkg-config (*.pc), установленные заданием. Для каждого модуля содержащее его задание фиксируется как поставщик модуля. Полученное сопоставление (module-to-package) сохраняется в переменной PKGDATA_DIR задачей do_packagedata для глобального доступа.

  Одновременно все устанавливаемые заданием модули pkg-config проверяются на предмет зависимости от других модулей pkg-config. Модуль считается зависимым от другого модуля, если он содержит строку "Requires:", указывающую другой модуль. Для каждой зависимости просматривается переменная PKGDATA_DIR для обнаружения пакетов, которые могут содержать модуль. При нахождении такого пакета добавляется зависимость во время работы от содержащего модуль пакета.

  Механизм pcdeps чаще всего определяет зависимости между пакетами -dev.

- *depchains.* Если пакет foo зависит от пакета bar, пакеты foo-dev и foo-dbg также зависят от bar-dev и bar-dbg, соответственно. Например, для -dev пакет bar-dev может предоставлять заголовки и символьные ссылки на общие библиотеки, требуемые foo-dev, что указывает на зависимость между этими пакетами. Зависимости, добавляемые depchains, имеют форму RRECOMMENDS.

  По умолчанию foo-dev имеет также зависимость RDEPENDS от foo, поскольку принятое по умолчанию значение RDEPENDS_${PN}-dev (в bitbake.conf) включает "${PN}".

  Для обеспечения непрерывности цепочек зависимостей пакеты -dev и -dbg всегда создаются по умолчанию, даже если результат будут пустым.

Задача do_package зависит от do_packagedata каждого задания в DEPENDS через декларацию [deptask], которая гарантирует, что требуемая информация о сопоставлениях зависимостей будет доступна при корректной установке DEPENDS.

4.7. Fakeroot и Pseudo

Некоторые задачи (например, do_install, do_package_write*, do_rootfs, do_image*) проще выполнить при наличии прав, которые обычно предоставляются пользователю root. Например, задаче do_install нужно задавать идентификаторы UID GID для файлов с произвольными значениями.

Одним из вариантов выполнения задач, доступных лишь пользователю root, является запуск BitBake от имени root. Однако этот способ громоздок и может создавать проблемы безопасности. Фактически используемый подход заключается в запуске задач, которым нужны полномочия root, в фиктивной (fake) среде root, где задача и её дочерние процессы считают, что они работают от имени root и получают непротиворечивое представление о файловой системе. До тех пор, пока для генерации финального вывода (например, пакета или образа) не требуются полномочия root, выполнение некоторых предшествующих этапов не вызывает проблем.

Возможность запуска задач в фиктивной среде root называют fakeroot от ключевого слова (флага переменной) BitBake, запрашивающего для задачи фиктивную среду root.

В системе сборки OE программа, реализующая fakeroot, называется Pseudo. Эта программа переопределяет системные вызовы, используя переменную окружения LD_PRELOAD, что создает иллюзию работы от имени root. Для отслеживания фиктивного владения и прав доступа к файлам в результате операций, требующих полномочий root, Pseudo использует базу данных SQLite 3, которая хранится в файлах ${WORKDIR}/pseudo/files.db для отдельных заданий. Хранение базы в файле, а не в памяти, обеспечивает сохранение данных между разными задачами и сборками, что недоступно при использовании fakeroot.

При добавлении задачи, работающей с теми же файлами и каталогами, что и задача fakeroot, её также нужно запускать в режиме fakeroot. В противном случае задача не сможет выполнить операции, разрешённые только пользователю root, и не увидит принадлежности и прав доступа для файлов, установленных другими задачами. Нужно также добавить зависимость от virtual/fakeroot-native:do_populate_sysroot, как показано ниже.

```
fakeroot do_mytask () {
    ...
}
do_mytask[depends] += "virtual/fakeroot-native:do_populate_sysroot"
```

Дополнительная информация приведена в описании переменных FAKEROOT* [7]. Базовые сведения о работе с Fakeroot и Pseudo даны в статье [8].

Литература

[1] Yocto Project Development Tasks Manual, http://www.yoctoproject.org/docs/3.0/dev-manual/dev-manual.html (перевод).

[2] Yocto Project Application Development and the Extensible Software Development Kit (eSDK), http://www.yoctoproject.org/docs/3.0/sdk-manual/sdk-manual.html (перевод).

[3] Yocto Project Reference Manual, http://www.yoctoproject.org/docs/3.0/ref-manual/ref-manual.html (перевод).

[4] Yocto Project Board Support Package (BSP) Developer's Guide, http://www.yoctoproject.org/docs/3.0/bsp-guide/bsp-guide.html (перевод).

[5] Yocto Project Quick Build, http://www.yoctoproject.org/docs/3.0/brief-yoctoprojectqs/brief-yoctoprojectqs.html.

[6] Toaster User Manual, http://www.yoctoproject.org/docs/3.0/toaster-manual/toaster-manual.html.

[7] BitBake User Manual, https://www.yoctoproject.org/docs/3.0/bitbake-user-manual/bitbake-user-manual.html (перевод).

[8] Why Not Fakeroot?, https://github.com/wrpseudo/pseudo/wiki/WhyNotFakeroot.

[9] Yocto Project Linux Kernel Development Manual, http://www.yoctoproject.org/docs/3.0/kernel-dev/kernel-dev.html (перевод).

[10] Pro Git, https://github.com/progit/progit2-ru/releases/download/2.1.22/progit_v2.1.22.pdf.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru