Разработка приложений Yocto Project и eSDK

Scott Rifenbark

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

Оглавление

Глава 1. Начальные сведения

1.1. Ведение

В этом документе описано использование стандартных и расширяемых SDK[1] Yocto Project (YP) для разработки приложений и образов. До выпуска YP 2.0 разработка приложений выполнялась с использованием ADT[2], инструментов кросс-разработки и других средств. В выпуске YP 2.1 разработка была перенесена в расширяемый SDK с большим набором инструментов и стандартный SDK. Все варианты SDK включают:

- *инструменты кросс-разработки* - компилятор, отладчик и другие инструменты;
- *библиотеки, заголовочные файлы и символы* для конкретных образов;
- *сценарий организации среды* - файлы *.sh, запускаемые однократно для организации среды кросс-разработки путём определения переменных и подготовки SDK к работе.

Расширяемый SDK дополнительно включает инструменты, упрощающие добавление в образ новых приложений и библиотек, изменение источников для имеющихся компонент, тестирование образов на целевых устройствах, а также для интеграции приложений в систему сборки OpenEmbedded (OE).

Можно применять SDK для независимой разработки и тестирования кода, предназначенного для целевой машины. SDK являются самодостаточными, двоичные файлы компонуются со своей копией libc, что обеспечивает независимость от целевой системы. Для достижения этого настраивается указатель на динамический загрузчик во время установки, поскольку путь не может быть изменён динамически. Это является причиной создания архивов populate_sdk и populate_sdk_ext.

Другим свойством SDK является создание единственного набора двоичных инструментов кросс-компиляции для данной архитектуры. Это основано на том, что целевое оборудование может быть указано компилятору gcc как набор опций, устанавливаемых сценарием настройки среды и содержащих такие переменные, как CC и LD. Такой подход снижает размер инструментария, однако следует понимать, что для каждой цели все равно требуется sysroot, поскольку двоичные файлы зависят от платформы.

Среда разработки SDK содержит две основных компоненты.

- Автономный SDK, содержащий инструменты кросс-разработки для конкретной архитектуры и соответствующие sysroot (для себя и целевой платформы), созданные системой сборки OE (например, SDK). Инструменты и sysroot базируются на метаданных конфигурации и расширений, что обеспечивает возможность кросс-разработки на хосте для целевой платформы. Расширяемый SDK содержит также инструмент devtool.
- Эмулятор QEMU[3], позволяющий имитировать целевое оборудование. Строго говоря, QEMU не является частью SDK и эмулятор нужно собирать и включать отдельно. Однако он играет важную роль в процессе разработки.

Стандартный и расширяемый SDK имеют много общего, однако в eSDK включены мощные средства разработки. В таблице можно увидеть основные характеристики обоих вариантов.

Свойство	***Стандартный SDK***	***Расширяемый SDK***
Инструменты	+	+[4]
Отладчик	+	+[4]
Размер	От 100 Мбайт	От 1 Гбайт (от 300 Мбайт в сокращённом варианте)
devtool	-	+
Сборка образов	-	+
Обновляемость	-	+
Управление Sysroot[5]	-	+
Установленные пакеты	-[6]	+[7]
Результат	Пакеты	Общее состояние

1.1.1. Инструменты кросс-разработки

Инструментарий кросс-разработки включает кросс-компилятор, кросс-компоновщик и кросс-отладчик, которые применяются при разработке пользовательских приложений для целевого оборудования. Расширяемый SDK включает также программу devtool. Инструментарий создаётся сценарием установки SDK или через каталог сборки с конфигурацией метаданных и расширений для целевого устройства. Кросс-инструменты работают с соответствующей файловой системой sysroot.

1.1.2. Sysroot

Естественная и целевая система sysroot содержат заголовочные файлы и библиотеки для создания двоичных файлов, работающих на целевой архитектуре. Целевая система sysroot основана на образе корневой файловой системы целевой платформы, создаваемом системой сборки OE с использованием тех же конфигурации метаданных, что и при сборке кросс-инструментов.

[1]Software Development Kit - пакет для разработки приложений.
[2]Application Development Toolkit - инструменты для разработки приложений.
[3]Quick EMUlator.
[4]eSDK содержит инструменты и отладчик, если SDK_EXT_TYPE = "full" или SDK_INCLUDE_TOOLCHAIN = "1".
[5]Управление sysroot осуществляется с помощью devtool, что снижает вероятность повреждений при добавлении библиотек.
[6]Управление пакетами при работе можно добавить в стандартный SDK (по умолчанию его нет).
[7]Нужно собрать и сделать доступным общее состояние для устанавливаемых пакетов.

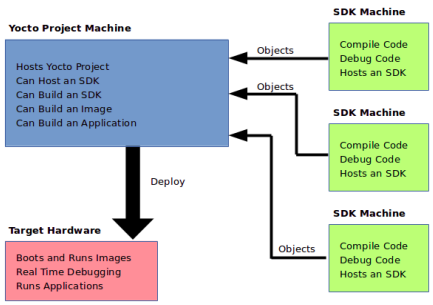
1.1.3. Эмулятор QEMU

Эмулятор QEMU позволяет имитировать оборудования для запуска образа или приложения. QEMU не является частью SDK и доступен несколькими способами:

- клонирование репозитория poky Git для создания дерева источников;
- загрузка и распаковка выпуска YP для создания дерево источников;
- установка архива кросс инструментов.

1.2. Модель разработки в SDK

Процесс разработки с использованием SDK показан на рисунке.



SDK устанавливается на любую машину и может применяться для разработки приложений, образов и ядер. Важно то, что машину с SDK не требуется связывать с машиной YP. Разработчик может независимо компилировать и тестировать объекты на своей машине, а по готовности включать их в образ, просто делая доступными для машины YP. Когда объект становится доступным, образ можно собрать заново с использованием YP.

Для работы нужно выполнить перечисленные ниже действия.

1. *Установка SDK для целевого оборудования*, как описано в параграфе 3.2. Установка SDK.
2. *Загрузка или сборка целевого образа.* YP поддерживает разную архитектуру и имеет множество собранных образов ядер и корневых файловых систем. Для разработки приложений под целевое оборудование следует выбрать целевую машину в разделе machines и загрузить нужные образы ядра и корневой файловой системы.

   При разработке приложений в запуском в эмуляторе QEMU следует выбрать область machines/qemu и найти там файлы для целевой архитектуры (например, qemux86_64 для 45-битовых платформ Intel®). Для использования корневой файловой системы в QEMU её потребуется распаковать (A.3. Извлечение корневой файловой системы.
3. *Разработка и тестирование приложения.* После выполнения предыдущих пунктов можно начинать разработку. Если нужно отдельно установить QEMU, можно найти нужный файл на сайте QEMU. Работа с эмулятором описана в разделе Using the Quick EMUlator (QEMU) [1].

Глава 2. Использование eSDK

В этой главе описана установка расширяемого SDK и работа с ним, а также функции devtool. Расширяемый SDK упрощает добавление в образ приложений и библиотек, менять источники имеющихся компонент, тестировать изменения на целевой платформе, а также упрощает интеграцию с системой сборки OE. Сравнение возможностей стандартного и расширяемого SDK приведено в разделе 1.1. Ведение.

В дополнение к функциональности devtool можно использовать инструменты напрямую (Глава 4. Использование инструментов SDK).

2.1. Что такое eSDK?

Расширяемый SDK предоставляет инструменты кросс-разработки и библиотеки, адаптированные к содержимому конкретного образа. Дополнением является мощный инструмент devtool, адаптированный для среды YP. Установленный eSDK включает множество файлов и каталогов, а также сценарий организации среды, конфигурационные файлы, встроенную систему сборки и devtool.

2.2. Установка eSDK

Первым делом нужно установить SDK на сборочный хост с помощью сценария *.sh. Можно загрузить архив установщика, включающий собранные инструменты, сценарий runqemu, систему сборки, devtool, и файлы поддержки из подходящего каталога toolchain в списке выпусков. Инструменты доступны для 32- и 64-битовой архитектуры. Инструменты YP основаны на образах core-image-sato и core-image-minimal и включают библиотеки для разработки.

Имена архивов с установочными сценариями представляют сначала хост-систему, а затем следует строка представления целевой архитектуры. Для eSDK в имени сценария присутствует -ext. Базовый формат имеет вид poky-glibc-host_system-image_type-arch-toolchain-ext-release_version.sh, где строка host_system указывает систему разработки (i686 или x86_64), image_type указывает образ, для которого будет собран SDK (core-image-sato или core-image-minimal), arch задаёт целевую архитектуру (aarch64, armv5e, core2-64, i586, mips32r2, mips64, ppc7400 или cortexa8hf-neon), release_version - номер выпуска YP (3.0, 3.0+snapshot). Например, установщик для 64-битовой хост-системы и целевой архитектуры i586-tuned на основе образа core-image-sato и текущего выпуска 3.0 будет называться poky-glibc-x86_64-core-image-sato-i586-toolchain-ext-3.0.sh.

Можно также загрузить SDK и собрать установщик, как описано в приложении A.2. Сборка установщика SDK.

SDK и инструменты самодостаточны ии устанавливаются в каталог poky_sdk внутри домашнего каталога, хотя можно выбрать иное место для установки. Однако местоположение SDK должна быть доступно для записи всем пользователям SDK. Ниже приведена команда для запуска установщика на 64-битовой системе x86 для 64-битовой архитектуры x86 с установкой SDK в каталог ~/Downloads/. Если у вас нет полномочий записи в каталог установки SDK, программа сообщит об этом и завершит работу.

```
$ ./Downloads/poky-glibc-x86_64-core-image-minimal-core2-64-toolchain-ext-2.5.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 2.5
==========================================================================
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "/home/scottrif/poky_sdk". Proceed [Y/n]? Y
Extracting SDK.............done
Setting it up...
Extracting buildtools...
Preparing build system...
Parsing recipes: 100% |#################################################################| Time: 0:00:52
Initialising tasks: 100% |##############################################################| Time: 0:00:00
Checking sstate mirror object availability: 100% |######################################| Time: 0:00:00
Loading cache: 100% |###################################################################| Time: 0:00:00
Initialising tasks: 100% |##############################################################| Time: 0:00:00
done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /home/scottrif/poky_sdk/environment-setup-core2-64-poky-linux
```

2.3. Запуск сценария организации среды eSDK

После установки SDK нужно запустить сценарий организации среды SDK, чтобы можно было начать работу. Сценарий размещается в каталоге, указанном при установке SDK (по умолчанию poky_sdk). Перед запуском сценария следует проверить его соответствие архитектуре, для которой планируется разработка. Имена сценариев организации среды начинаются с environment-setup и включают целевую архитектуру. Ниже приведен пример запуска сценария для целевой машины IA с архитектурой i586.

```
$ cd /home/scottrif/poky_sdk
$ source environment-setup-core2-64-poky-linux
 SDK environment now set up; additionally you may now run devtool to perform development tasks.
 Run devtool --help for further details.
```

Сценарий определяет переменные окружения для работы SDK (например, PATH, CC, LD и т. п.), список которых можно увидеть в файле сценария.

2.4. Использование devtool в процессах SDK

Одним из основных инструментов eSDK является команда devtool, обеспечивающая возможности сборки, тестирования и подготовки пакетов для программ в eSDK, а также их встраивание в образы, создаваемые системой сборки OE. Применение devtool не ограничивается eSDK и программу можно использовать для разработки любых проектов, вывод которых должен стать частью образа в системе сборки.

Команды devtool организационно похожи на команды Git, посмотреть список команд можно с помощью devtool --help. Подробная информация о devtool приведена в разделе devtool Quick Reference [2]. Тремя основными командами devtool для начала работы служат:

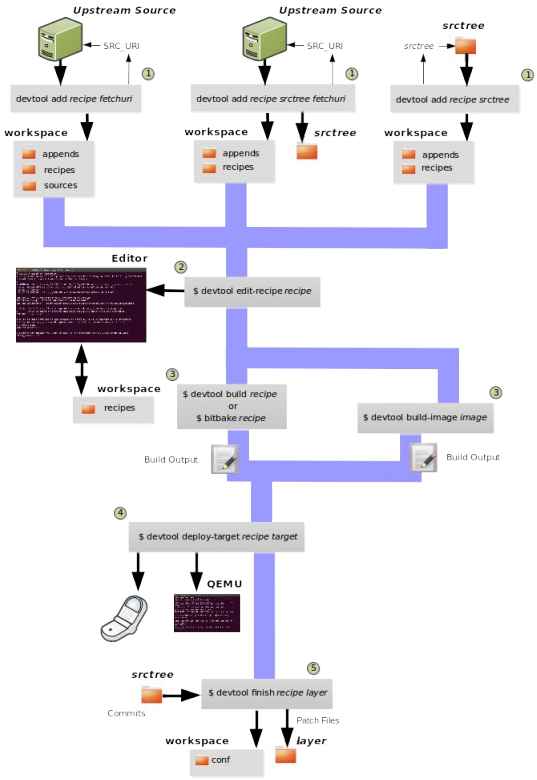
- devtool add - добавляет в сборку новую программу;
- devtool modify - устанавливает среду для изменения источников имеющихся компонент;
- devtool upgrade - обновляет имеющееся задание для сборки с обновлёнными исходными файлами.

Как и в системе сборки, задания представляют программные пакеты в devtool. Команда devtool add автоматически создает задание, с devtool modify используется имеющееся задание для указания местоположения источников и способа их изменения (patch). В обоих случаях среда выполнения организуется так, чтобы при сборке задания использовалось контролируемое дерево источников, что позволяет вносить нужные изменения в код. По умолчанию новые задания и исходный код помещаются в каталог workspace в дереве SDK.

2.4.1. Добавление приложения с помощью devtool add

Команда devtool add создает новое задание на основе имеющихся исходных кодов, используя уровень workspace, как и многие другие команды devtool. Команда достаточно гибко позволяет извлекать исходный код в workspace или локальный репозиторий Git, а также использовать имеющийся код, извлекать который не нужно. В зависимости от конкретной ситуации применяются разные аргументы и опции devtool add, как описано ниже и показано на рисунке.

1. *Генерация нового задания.* В общей среде разработки участники процесса обычно отвечают за свои области исходного кода. Для работы нужен доступ к коду, заданию и рабочей области. В верхней части рисунка показано 3 варианта использования devtool add для генерации задания на основе имеющихся источников.
    - *Левый вариант* представляет ситуацию где исходные коды не присутствуют локально и их нужно извлечь. Здесь извлечение кода выполняется в workspace. Команда devtool add recipe fetchuri извлечёт файлы исходного кода в локальный репозиторий Git в каталоге sources, затем создаст в workspace задание и файл добавления с указанным именем. Если параметр recipe не указан, команда попытается самостоятельно определить имя задания.
    - *Средний вариант* представляет случай, когда исходные коды не присутствуют локально и их нужно извлечь, но разместить за пределами локального каталога workspace. Команда devtool add recipe srctree fetchuri создаст локальный репозиторий Git в каталоге, указанном параметром srctree. Также будут созданы файлы задания (recipe) и связанного с ним дополнения.
    - *Правый вариант* соответствует случаю, когда исходные коды уже имеются за пределами workspace. Команда devtool add recipe srctree проверит исходный код в srctree и создаст файлы задания (recipe) и дополнения в workspace.



2. *Редактирование задания* с помощью команды edit-recipe позволяет использовать редактор, указанный в переменной окружения $EDITOR.
3. *Сборка задания или повторная сборка образа* выполняется с помощью команды devtool build recipe или devtool build-image image.
4. *Развёртывание результатов сборки* после команды devtool build для проверки работоспособности на целевой системе или в эмуляторе QEMU. При развёртывании образа предполагается наличие в этом образе SSH, а при использовании реального оборудования - доступ к платформе через сеть с хоста разработки. Для развёртывания образа служит команда devtool deploy-target recipe target, где параметр target указывает целевую систему. Для переноса образа на физическую платформу потребуются дополнительные команды в зависимости от конкретного устройства.
5. *Завершение работы с заданием* по команде devtool finish recipe layer создает все нужные правки (patch) соответствующие фиксациям (commit) в локальном репозитории Git, переносит задание на постоянный уровень (layer) и настраивает задание для обычной сборки вместо использования workspace. Восстанавливаются также состояния стандартных уровней и дерева источников. При необходимости можно воспользоваться командой devtool reset для возврата к работе с заданием.

2.4.2. Изменение кода с помощью devtool modify

Команда devtool modify готовит способ работы с имеющимся кодом, для которого уже есть локальное задание для сборки программ. Команда позволяет извлекать код из восходящего источника, указывать имеющееся задание, а также отслеживать и извлекать любые файлы исправлений (patch) от других разработчиков, связанные с заданием. В зависимости от конкретной ситуации применяются разные аргументы и опции devtool modify, как описано ниже и показано на рисунке.

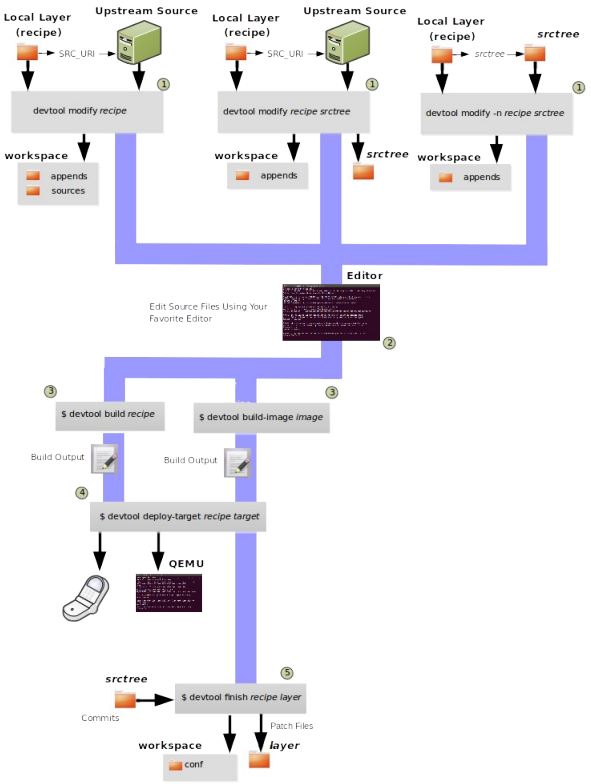
1. *Подготовка к изменению кода* показана в верхней части рисунка в предположении, что задание размещено локально на уровне, не входящем в workspace, а исходные файлы размещены в несжатом виде локально или в восходящем репозитории.

   - *Левый вариант* показывает случай, когда исходных кодов нет локально и нужно их извлечь. По умолчанию полученные файлы записываются в workspace. Задание размещается на своём уровне за пределами workspace. Команда devtool modify recipe находит задание и извлекает файлы исходного кода, используя переменную SRC_URI для поиска исходного кода и локальных исправлений (patch) от других разработчиков. Здесь не используется аргумент srctree, поэтому команда devtool modify по умолчанию извлекает файлы, указанные SRC_URI, в локальный репозиторий Git внутри workspace. В результате выполнения команды исходный код и файлы дополнения размещаются в workspace, а задание - на своём уровне.

     При наличии локальных файлов, не являющихся исправлениями (файлы, указанные с file:// в переменной SRC_URI, за исключением *.patch и *.diff) эти файлы копируются в каталог oe-local-files созданного дерева исходного кода, где эти файлы можно редактировать. Внесённые в эти файлы изменения или добавления будут включены в следующую сборку вместе с другими изменениями исходного кода.

   - *Средний вариант* показывает случай, когда исходных кодов нет локально и нужно их извлечь, но они помещаются в локальный репозиторий Git, указанный параметром srctree. Команда devtool modify

recipe srctree указывает изменяемое задание и локальный каталог для записи извлечённых файлов. В параметре srctree не допускается указание URL.



Переменная SRC_URI указывает местоположение всех локальных файлов, связанных с исправлениями (patch), остальные локальные файлы копируются в каталог oe-local-files созданного дерева кодов. Внутри workspace команда создает файл дополнения для задания, само задание сохраняется в исходном месте, а исходные коды помещаются в каталог, указанный srctree.

- *Правый вариант* относится к случаю, когда исходные коды уже имеются локально (srctree) в виде репозитория Git за пределами workspace. Задание размещается локально на своём уровне. Команда devtool modify -n recipe srctree использует опцию -n для указания того, что исходный код не нужно извлекать, а srctree задаёт его местоположение. Если каталог oe-local-files уже имеется в дереве кода и содержит файлы, не являющиеся правками (patch), эти файлы будут использованы. Однако если этого каталога нет и была использована команда devtool, эти файлы будут удалены.

  По завершении команды devtool modify будет создан лишь файл дополнения в workspace, а задание и исходные коды сохранятся на своих местах.

2. *Редактирование задания* по завершении команды devtool modify возможно с помощью любого редактора.
3. *Сборка задания или повторная сборка образа* выполняется с помощью команды devtool build recipe или devtool build-image image.
4. *Развёртывание результатов сборки после команды* devtool build для проверки работоспособности на целевой системе или в эмуляторе QEMU. При развёртывании образа предполагается наличие в этом образе SSH, а при использовании реального оборудования - доступ к платформе через сеть с хоста разработки. Для развёртывания образа служит команда devtool deploy-target recipe target, где параметр target указывает целевую систему. Для переноса образа на физическую платформу потребуются дополнительные команды в зависимости от конкретного устройства.
5. *Завершение работы с заданием* по команде devtool finish recipe layer создает все нужные правки (patch) соответствующие фиксациям (commit) в локальном репозитории Git, переносит задание на постоянный уровень (layer) и настраивает задание для обычной сборки вместо использования workspace. Восстанавливаются также состояния стандартных уровней и дерева источников. При необходимости можно воспользоваться командой devtool reset для возврата к работе с заданием.

2.4.3. Использование devtool upgrade для обновления задания

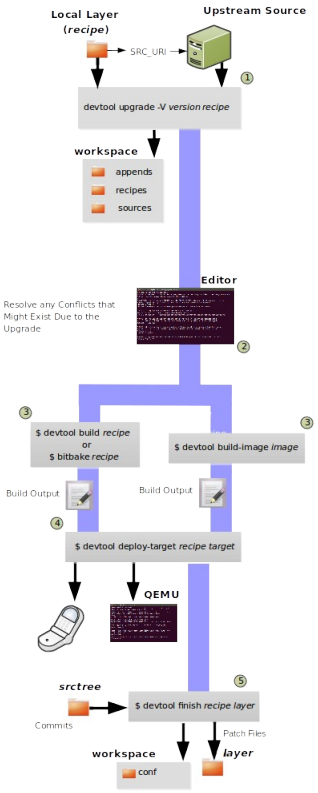
Команда devtool upgrade обновляет имеющееся задание путём использования более свежей версии исходного кода в восходящем репозитории. Команда является одним из вариантов обновления заданий, другие методы описаны в разделе Upgrading Recipes [1].

Команда devtool upgrade позволяет гибко задавать выпуск исходного кода и схемы версий, извлекать код в workspace или в отдельный каталог и может работать с любыми формами исходного кода, поддерживаемыми сборщиками.

Процесс обновления показан на рисунке и описан ниже.

1. *Инициирование обновления* с помощью команды devtool upgrade требует наличия локального уровня вне workspace, а также наличия исходных кодов в каталоге, указанном переменной SRC_URI в задании (например, архив с новой версией в имени или другой выпуск восходящего репозитория Git). Для обновления задания служит команда devtool upgrade -V version recipe. По умолчанию команда извлекает исходный код в каталог

sources внутри workspace. Для записи файлов в другой каталог следует использовать команду devtool upgrade -V version recipe srctree. Опция -V указывает новую версии, по умолчанию извлекается последняя версия.



Если исходные файлы, указанные SRC_URI, размещены в репозитории Git, нужно использовать опцию -S с указанием выпуска (revision) программы.

После того, как devtool найдёт задание, используется переменная SRC_URI для нахождения исходного кода и всех изменений (patch) от других разработчиков. В результате команда организует исходный код, новую версию задания, а также файл дополнения в workspace.

При наличии локальных файлов, не являющихся исправлениями (файлы, указанные с file:// в переменной SRC_URI, за исключением *.patch и *.diff) эти файлы копируются в каталог oe-local-files созданного дерева исходного кода, где эти файлы можно редактировать. Внесённые в эти файлы изменения или добавления будут включены в следующую сборку вместе с другими изменениями исходного кода.

2. *Устранение конфликтов, связанных с обновлением,* которые могут возникать в результате несоответствия изменений в правках (patch), указанных SRC_URI, новым версиям программ. В таких случаях нужно редактировать файлы исходного кода и использовать процедуру git rebase для устранения конфликта. Все конфликты должны быть устранены до перехода к следующему этапу.
3. *Сборка задания или повторная сборка образа* выполняется с помощью команды devtool build recipe или devtool build-image image.
4. *Развёртывание результатов сборки после команды* devtool build для проверки работоспособности на целевой системе или в эмуляторе QEMU. При развёртывании образа предполагается наличие в этом образе SSH, а при использовании реального оборудования - доступ к платформе через сеть с хоста разработки. Для развёртывания образа служит команда devtool deploy-target recipe target, где параметр target указывает целевую систему. Для переноса образа на физическую платформу потребуются дополнительные команды в зависимости от конкретного устройства.
5. *Завершение работы с заданием* по команде devtool finish recipe layer создает все нужные правки (patch) соответствующие фиксациям (commit) в локальном репозитории Git, переносит задание на постоянный уровень (layer) и настраивает задание для обычной сборки вместо использования workspace. Восстанавливаются также состояния стандартных уровней и дерева источников. При необходимости можно воспользоваться командой devtool reset для возврата к работе с заданием.

2.5. Инструмент devtool add

Команда devtool add автоматически создает задание на основе дерева исходных кодов, указанного в команде. В настоящее время команда способна работать с заданиями:

- Autotools (autoconf и automake);
- Cmake;
- Scons;
- qmake;
- Makefile;
- внешние модули ядра;

- двоичные пакеты (опция -b);
- модули Node.js;
- модули Python, использующие setuptools или distutils.

За исключением двоичных пакетов, определение способа обработки исходных кодов выполняется автоматически на основе имеющихся в дереве кода файлов. Например, при обнаружении файла CMakeLists.txt предполагается использование CMake.

В большинстве случаев для правильной сборки нужно редактировать созданное автоматически задание. Для получения нужного результата может потребоваться несколько итераций. После сборки можно проверить задание на целевом устройстве.

2.5.1. Имя и версия

Если в команде не указано имя и версия, devtool add использует метаданные дерева источников в попытке определения имени и версии собираемой программы. На основе этих данных указывается имя и создаётся задание. Если имя и версию определить не удалось, выводится сообщение об ошибке. В таких случаях следует повторить команду с указанием хотя бы имени или версии. Иногда определение имени и версии из дерева кода может быть ошибочным. В таких случаях следует сбросить задание командой devtool reset -n recipename, а затем снова ввести devtool add с указанием имени и/или версии.

2.5.2. Нахождение и отображение зависимостей

Команда devtool add пытается найти зависимости при сборке и отображает их на другие задания в системе, внося их имена в переменную DEPENDS. Если зависимость не удаётся сопоставить, выводится сообщение об этом. Невозможность отобразить зависимость может быть связана с тем, что имя не распознано или недоступно. В последнем случае следует использовать команду devtool add для добавления задания, выполняющего зависимость и после этого обновить переменную DEPENDS в исходном задании, включив в неё новое задание.

Если нужно добавить зависимости при работе, это можно сделать в форме RDEPENDS_${PN} += "dependency1 dependency2 …".

Команда devtool add зачастую не может различить обязательные и необязательные зависимости и некоторые найденные зависимости могут оказаться необязательными. В случае сомнений следует обратиться к документации или сценарию configure в программе собираемого задания. Иногда необязательную зависимость можно исключить параметром сценария configure.

2.5.3. Описание лицензии

Команда devtool add пытается определить, может ли программа распространяться по общепринятой лицензии для открытого кода и при положительном ответе устанавливает значение переменной LICENSE. Следует внимательно проверить добавленное в переменную значение по документации и файлам исходного кода собираемой программы и при необходимости скорректировать переменную LICENSE.

Команда devtool add устанавливает значение переменной LIC_FILES_CHKSUM с указанием всех файлов, относящихся к лицензии. Следует помнить, что информация о лицензии часто указывается в комментариях в начале исходных файлов или документации и команда в таких случаях не находит эти данные. Поэтому может потребоваться изменение переменной LIC_FILES_CHKSUM для указания таких комментариев. Особенно важна установка LIC_FILES_CHKSUM для сторонних программ. Механизм контрольных сумм пытается обеспечить корректное указание лицензий при обновлении задания. Все изменения файлов с лицензиями обнаруживаются и выводится сообщение об ошибке с предложением проверить текст лицензии.

Если команда devtool add не может найти данных о лицензировании, в переменной LICENSE устанавливается значение CLOSED, а LIC_FILES_CHKSUM сбрасывается. Это позволяет продолжить разработку, хотя установки вряд ли будут корректны во всех случаях. Следует проверить документацию и файлы исходного кода собираемой программы для определения действующей лицензии.

2.5.4. Добавление программ на основе Makefile

Инструмент Make очень часто применяется в фирменных и открытых программах, но файлы Makefile зачастую не рассчитаны на кросс-компиляцию. Поэтому devtool часто не может обеспечить корректную сборку с такими файлами. Например, часто вызывается напрямую компилятор gcc вместо использования переменной CC. В среде кросс-компиляции gcc обычно является компилятором сборочного хоста, а кросс-компилятор называется иначе, например, arm-poky-linux-gnueabi-gcc и может требовать дополнительных аргументов (например, sysroot для целевой машины).

При подготовке заданий для программ, собираемых только на основе Makefile следует учитывать ряд аспектов.

- Возможно придётся изменить Makefile для замены жёстко заданных инструментов сборки (скажем, gcc и g++).
- Среда для работы Make организуется с использованием различных стандартных переменных (например, CC, CXX и т. п.) аналогично организации среды сценарием настройки SDK. Простейшим способом увидеть эти переменные является ввод команды devtool build для задания и просмотр файла oe-logs/run.do_compile, в начале которого будет присутствовать список устанавливаемых переменных. Можно воспользоваться этими переменными в Makefile.
- Если в Makefile принятое по умолчанию значение переменной установлено оператором =, оно переопределяет установленное в среде значение, что обычно нежелательно. В этом случае можно заменить в Makefile оператор на ?= или принудительно установить значение в команде make. Для принудительной установки значений в командной строке следует добавить значения в переменную задания EXTRA_OEMAKE или PACKAGECONFIG_CONFARGS. Например, EXTRA_OEMAKE += "'CC=${CC}' 'CXX=${CXX}'". Здесь параметры даны в одинарных кавычках, поскольку значения могут включать пробелы.

- Жёсткое указание путей в Makefile зачастую создает проблемы при кросс-компиляции, поскольку эти пути обычно указывают каталоги хоста сборки, которые могут быть доступны лишь для чтения или приводить к «загрязнению» кросс-компиляции, поскольку относятся к хосту сборки, а не к целевой системе. Следует исправить Makefile, используя переменные с префиксом или иные переменные для решения проблемы.
- Иногда Makefile использует относящиеся к цели команды, такие как ldconfig. В таких случаях можно применить правки, удаляющие ненужные команды из Makefile.

2.5.5. Добавление естественных инструментов

Часто требуется собрать дополнительные инструменты для работы на хосте сборки, а не в целевой системе. Следует указывать такие требования при запуске devtool add одним из указанных ниже способов.

- Указать имя задания с суффиксом -native для сборки его лишь на сборочном хосте.
- Указать опцию --also-native в команде devtool add command, создающую дополнительно файл задания -native.

Если нужно добавить инструмент, включенный в дерево кода, собираемого для цели, можно собрать компоненты для хоста сборки и цели раздельно, а не в одном процессе компиляции, например с помощью опции --also-native.

2.5.6. Добавление модулей Node.js

Команду devtool add можно применять для добавления модулей Node.js через npm, а также из удалённого или локального репозитория. Добавление через npm выполняется по команде вида devtool add "npm://registry.npmjs.org;name=forever;version=0.15.1". Параметры name и version являются обязательными. Создаются файлы блокировки и упаковки, которые указываются в задании для «замораживания» выбранной изначально версии в зависимостях. Сохраняются также контрольные суммы для проверки при последующих выборках. Это обеспечивает воспроизводимость и целостность задания.

URL нужно указывать в кавычках. Это не нужно devtool add, но оболочка считает символ ; разделителем команд, поэтому без кавычек devtool add не получит другие части и возникнет ошибка «command not found». Для поддержки добавления модулей Node.js в SDK нужно включить задание nodejs.

При добавлении модулей Node.js из локального или удалённого репозитория используется команда вида devtool add https://github.com/diversario/node-ssdp. В этом случае devtool извлечёт файлы из репозитория Git, определит код Node.js, извлечёт файлы с помощью npm и установит переменную SRC_URI.

2.6. Работа с заданиями

При сборке задания с помощью команды devtool build обычно выполняются перечисленные ниже действия.

1. Извлечение исходного кода.
2. Распаковка исходного кода.
3. Настройка конфигурации.
4. Компиляция.
5. Установка результатов сборки.
6. Упаковка установленных результатов (создание пакета).

Для заданий из рабочего пространства извлечение и распаковка исходного кода отключены, поскольку код уже подготовлен к использованию. Каждый из этапов сборки оформлен в виде функции (задачи), которая обычно имеет префикс do_ (например, do_fetch, do_unpack). Эти функции обычно являются сценариями оболочки, но могут быть написаны и на языке Python.

При просмотре содержимого задания можно увидеть, что оно не включает полных инструкций по сборке программ. Общая функциональность размещена в классах, наследуемых с помощью директивы inherit. Этот метод позволяет указывать в задании лишь специфические аспекты собираемой программы. Класс base неявно наследуется всеми заданиями и обеспечивает функциональность, требуемую для большинства заданий.

2.6.1. Поиск журнала и рабочих файлов

После первого вызова команды devtool build задания, созданные с помощью команды devtool add, чей исходный код был изменён командой devtool modify, содержат в дереве исходного кода символьные ссылки, указанные ниже.

- *oe-logs* указывает каталог, где создаются журнальные файлы и сценарии запуска для каждого этапа сборки.
- *oe-workdir* указывает временную рабочую область задания. В этой области важны несколько каталогов.
  - *image/* содержит все файлы, установленные задачей do_install. В задании этот каталог обозначен ${D}.
  - *sysroot-destdir/* содержит часть файлов, установленных задачей do_install, которые помещены в общий каталог sysroot (2.6.3. Совместное использование файлов заданиями).
  - *packages-split/* содержит каталоги для каждого пакета, созданного заданием (2.6.4. Подготовка пакетов).

Информация в этих каталогах поможет разобраться с каждым этапом сборки.

2.6.2. Установка параметров настройки

Если программы задания собираются с помощью GNU autoconf, программе передаётся фиксированный набор аргументов для включения кросс-компиляции и дополнений, заданных переменной EXTRA_OECONF или PACKAGECONFIG_CONFARGS в задании. Если нужно передать дополнительные параметры, их следует добавлять в переменную EXTRA_OECONF или PACKAGECONFIG_CONFARGS. В других поддерживаемых инструментах сборки применяются похожие переменные (например, EXTRA_OECMAKE для CMake, EXTRA_OESCONS для Scons и т. п.).

Если нужно передать опции команде make, можно использовать переменную EXTRA_OEMAKE или PACKAGECONFIG_CONFARGS.

Для получения информации об установке упомянутых выше параметров можно воспользоваться командой devtool configure-help. Команда точно определяет передаваемые опции и показывает их вместе с другими аргументами, которые могут быть переданы через EXTRA_OECONF или PACKAGECONFIG_CONFARGS. Если это возможно, команда также покажет вывод сценария configure с опцией --help в качестве справки.

2.6.3. Совместное использование файлов заданиями

Задания часто используют файлы других заданий на хосте сборки. Например, приложению, связанному с общей библиотекой, нужно доступ к библиотеке и соответствующим заголовкам. Такой доступ обеспечивается в eSDK через sysroot. Каталог имеется для каждой «машины» поддерживаемой SDK. На практике это означает наличие sysroot для целевой машины и хоста сборки.

Заданиям никогда не следует записывать файлы напрямую в sysroot, а нужно устанавливать их с помощью задачи do_install в стандартные места внутри каталога ${D}. Часть этих файлов автоматически попадает в sysroot. Такое ограничение обусловлено тем, что практически все файлы sysroot указываются в манифестах, чтобы обеспечить возможность их удаления при изменении или удалении задания. Это позволяет исключать устаревшие файлы.

2.6.4. Подготовка пакетов

Подготовка пакетов не всегда актуальная для eSDK. Однако при рассмотрении способов включения вывода сборки в целевой образ важно понимать процесс подготовки пакетов, поскольку именно они, а не задания включаются в образ.

При выполнении задачи do_package файлы, установленные задачей do_install делятся на основной пакет, который почти всегда называется по имени задания, и несколько других пакетов. Это разделение обусловлено тем, что не все установленные файлы нужны в образе. Например, в рабочем образе может быть не нужна документация, для каждого задания файлы документации выделяются в пакет -doc. Задания для пакетов с необязательными модулями могут подвергаться дополнительному разделению.

После сборки задания можно посмотреть распределение файлов в каталоге oe-workdir/packages-split, где каждый пакет помещён в свой каталог. Распределение файлов определяется переменными PACKAGES и FILES за исключением некоторых сложных случаев. В переменной PACKAGES указаны все созданные пакеты, а FILES содержит распределение файлов по пакетам с использованием переопределений для указания пакетов. Например, FILES_${PN} указывает файлы основного пакета (который называется по имени задания ${PN}). Порядок значений в переменной PACKAGES важен и для каждого установленного файла первый пакет, значение для которого в FILES соответствует файлу, будет пакетов, куда включается файл. Для переменных PACKAGES и FILES имеются принятые по умолчанию значения, поэтому их установка в задании может оказаться ненужной, если задание применяет стандартное размещение файлов.

2.7. Восстановление исходного состояния целевого устройства

При использовании команды devtool deploy-target для записи вывода задания в целевую систему и работе с имеющимися компонентами системы может возникнуть потребность восстановить файлы, которые имелись до ввода команды devtool deploy-target. Эта команда создает копии перезаписываемых файлов и можно воспользоваться командой devtool undeploy-target для восстановления, например, devtool undeploy-target lighttpd root@192.168.7.2. При развёртывании нескольких приложений можно отказаться от всех с помощью опции -a, например, devtool undeploy-target -a root@192.168.7.2.

Информация о развёрнутых файлах , а также сохранённых копиях размещается в целевой системе, для чего на машине требуется дополнительное пространство.

Команды devtool deploy-target и devtool undeploy-target в настоящее время не взаимодействуют с системами управления пакетами на целевом устройстве (например, RPM или OPKG). Поэтому не следует смешивать на целевом устройстве операции менеджера пакетов и devtool deploy-target во избежание конфликтов.

2.8. Установка дополнительных элементов в eSDK

Пакет eSDK обычно распространяется с небольшим набором инструментов и библиотек. Минимальный вариант SDK является почти пустым и заполняется по мере необходимости. Иногда может явно требоваться установка дополнительных элементов в SDK. Такие элементы следует сначала найти с помощью команды devtool search. Предположим, например, что нужна привязка к libGL, но пакет с libGL не известен. Можно найти его по команде

```
$ devtool search libGL
mesa                    A free implementation of the OpenGL API
```

Узнав задание (mesa), его можно установить командой вида devtool sdk-install mesa. По умолчанию команда devtool sdk-install предполагает наличие собранного задания у поставщика SDK. Если элемент недоступен и его можно собрать из исходного кода, можно использовать опцию -s.

```
$ devtool sdk-install -s mesa
```

Важно помнить, что сборка элемента занимает существенно больше времени, нежели установка собранного. Если задания для элемента нет, но нужно добавить элемент в SDK, сначала потребуется команда devtool add.

2.9. Обновление установленного eSDK

При работе с eSDK, который время от времени обновляется (сторонний SDK), может потребоваться установка обновлений SDK вручную. Для этого служит команда devtool sdk-update. Здесь предполагается, что у поставщика SDK имеется принятый по умолчанию идентификатор URL для обновления (SDK_UPDATE_URL), как указано в приложении B.4. Обновление eSDK после установки. При отсутствии такого URL, его нужно указать явно в виде devtool sdk-update path_to_update_directory, указывая URL опубликованного SDK, а не установщика, который был загружен ранее.

2.10. Создание SDK с дополнительными компонентами

Может потребоваться включение в SDK своих библиотек. Например, клиенты могут использовать ваш SDK для сборки своих приложений и им потребуются специальные библиотеки. В этом случае можно создать SDK на базе имеющейся инсталляции, выполнив несколько шагов.

1. Установка при необходимости eSDK в качестве базы для производного SDK.
2. Выполнение сценария организации среды для SDK.
3. Добавление библиотек и других компонент с помощью команды devtool add.
4. Выполнение команды devtool build-sdk.

На этих этапах принимаются задания, добавленные в workspace и создаётся новый установщик SDK, содержащий эти задания и соответствующие двоичные файлы. Задания переносятся в отдельный уровень производного SDK, оставляя каталог workspace чистым и готовым для работы с другими заданиями.

Глава 3. Использование стандартного SDK

В этой главе описана установка и использование стандартного SDK. Сравнение стандартных и расширяемых SDK приведено в разделе 1.1. Ведение. Стандартный SDK позволяет работать с проектами на основе Makefile и Autotools. Глава 4. Использование инструментов SDK содержит более подробные сведения.

3.1. Что такое стандартный SDK?

Стандартный SDK включает инструменты и библиотеки для кросс-разработки, адаптированные под конкретный образ. Стандартный SDK является более традиционным средством разработки, нежели eSDK, с дополнительным инструментом devtool. Установленный SDK включает множество файлов и каталогов, а также сценарий организации среды, конфигурационные файлы, корневые файловые системы для хоста и цели. Структура каталогов описана в приложении A.4. Структура каталогов стандартного SDK.

3.2. Установка SDK

Первым делом нужно установить SDK на хосте сборки с помощью установочного сценария *.sh. Можно загрузить архив установщика, включающий собранные инструменты и файлы поддержки, из подходящего каталога toolchain в Index of Releases. Инструменты доступны для 32- и 64-битовой архитектуры. Инструментарий YP основан на образе core-image-sato или core-image-minimal и включает нужные для разработки библиотеки.

Имена архивов установщиков начинаются с представления хост-системы, за которой указывается целевая архитектура. Например, poky-glibc-host_system-image_type-arch-toolchain-release_version.sh, где строка host_system указывает систему разработки (i686 или x86_64), image_type указывает образ, для которого будет собран SDK (core-image-sato или core-image-minimal), arch задаёт целевую архитектуру (aarch64, armv5e, core2-64, i586, mips32r2, mips64, ppc7400 или cortexa8hf-neon), release_version - номер выпуска YP (3.0, 3.0+snapshot). Например, установщик для 64-битовой хост-системы и целевой архитектуры i586-tuned на основе образа core-image-sato и текущего выпуска 3.0 будет называться poky-glibc-x86_64-core-image-sato-i586-toolchain-3.0.sh.

Другим вариантом является самостоятельная сборка SDK, как описано в приложении A.2. Сборка установщика SDK.

SDK и инструменты самодостаточны ии устанавливаются в каталог poky_sdk внутри домашнего каталога, хотя можно выбрать иное место для установки. Однако местоположение SDK должна быть доступно для записи всем пользователям SDK. Ниже приведена команда для запуска установщика на 64-битовой системе x86 для 64-битовой архитектуры x86 с установкой SDK в каталог ~/Downloads/. Если у вас нет полномочий записи в каталог установки SDK, программа сообщит об этом и завершит работу.

```
  $ ./Downloads/poky-glibc-x86_64-core-image-sato-i586-toolchain-3.0.sh
Poky (Yocto Project Reference Distro) SDK installer version 3.0
===============================================================
Enter target directory for SDK (default: /opt/poky/3.0):
You are about to install the SDK to "/opt/poky/3.0". Proceed [Y/n]? Y
Extracting SDK.....................................  ..............................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
 $ . /opt/poky/3.0/environment-setup-i586-poky-linux
```

В приложении A.4. Структура каталогов стандартного SDK описаны каталоги установленного SDK.

3.3. Запуск сценария настройки среды SDK

После установки SDK нужно запустить сценарий организации среды SDK, чтобы можно было начать работу. Сценарий размещается в каталоге, указанном при установке SDK (по умолчанию /opt/poky/3.0). Перед запуском сценария следует проверить его соответствие архитектуре, для которой планируется разработка. Имена сценариев организации среды начинаются с environment-setup и включают целевую архитектуру. Ниже приведен пример запуска сценария для целевой машины IA с архитектурой i586.

```
$ source /opt/poky/3.0/environment-setup-i586-poky-linux
```

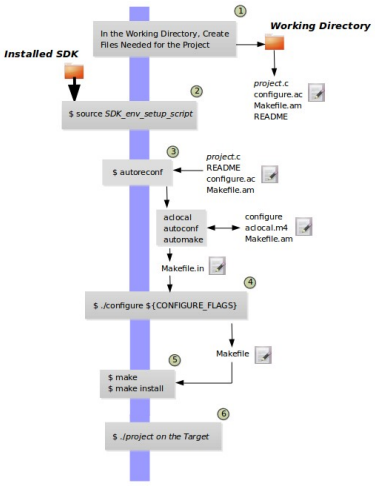
При запуске сценария установки задаются те же переменные окружения, что и для eSDK, описанные в разделе 2.3. Запуск сценария организации среды eSDK.

Глава 4. Использование инструментов SDK

Инструменты SDK можно применять напрямую в проектах на основе Makefile и Autotools.

4.1. Проекты на основе Autotools

После установки подходящих инструментов кросс-разработки можно начать работу на основе процесса GNU Autotools, не входящего в систему сборки OE. Этапы процесс показаны на рисунке, дополнительную информацию можно найти на сайте GNOME Developer.



Ниже приведен пример разработки на основе Autotools простого проекта Hello World.

1. *Создание и наполнение рабочего каталога*.

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

В рабочем каталоге нужно создать файл с исходным кодом проекта, файл для подготовки конфигурации, а также файл для создания Makefile и файл README - hello.c, configure.ac, Makefile.am, README.

Для создания файла README, требуемого GNU Coding Standards, служит команда touch README. Остальные файлы проекта показаны ниже.

- hello.c

```
#include <stdio.h>

main()
   {
      printf("Hello World!\n");
   }
```

- configure.ac

```
AC_INIT(hello,0.1)
AM_INIT_AUTOMAKE([foreign])
AC_PROG_CC
AC_CONFIG_FILES(Makefile)
AC_OUTPUT
```

- Makefile.am

```
bin_PROGRAMS = hello
hello_SOURCES = hello.c
```

2. *Выполнение сценария организации среды кросс-разработки* из каталога SDK. Имя сценария начинается с environment-setup и включает архитектуру машины, а затем строку poky-linux. Например, сценарий из принятой по умолчанию установки SDK для 32-битовой архитектуры Intel x86 и выпуска Zeus YP запускается командой source /opt/poky/3.0/environment-setup-i586-poky-linux.

3. *Создание сценария configure* с помощью команды autoreconf, которая запустит нужные инструменты Autotools, такие как aclocal, autoconf, automake. Если при работе autoreconf возникнут ошибки, связанные с configure.ac и указывающие отсутствие файлов, можно использовать опцию -i для копирования нужных файлов.

4. *Кросс-компиляция проекта* с использованием кросс-компилятора. Переменная окружения CONFIGURE_FLAGS задаёт минимальные аргументы для GNU configure

```
$ ./configure ${CONFIGURE_FLAGS}
```

Для проектов на основе Autotools можно использовать инструменты кросс-разработки, просто передав подходящую опцию host сценарию configure. Используемая опция host выводится из имени сценария организации среды в каталоге, куда установлены кросс-инструменты. Например, опция host для целевой платформы ARM, использующей GNU EABI, будет иметь значение armv5te-poky-linux-gnueabi (имя сценария environment-setup-armv5te-poky-linux-gnueabi). Приведённая ниже команда обновит проект и настроит его для сборки с помощью подходящих кросс-инструментов.

```
$ ./configure --host=armv5te-poky-linux-gnueabi --with-libtool-sysroot=sysroot_dir
```

5. *Сборка и установка проекта* с помощью приведённых ниже команд с указанием целевого каталога.

```
$ make
$ make install DESTDIR=./tmp
```

Переменные окружения, используемые сценарием организации среды кросс-разработки и их переопределение в Makefile описаны в разделе 4.2. Задания на основе Makefile.

Следующая команда обеспечивает простой способ проверить инсталляцию проекта, выводя архитектуру, где может работать двоичный файл (должна совпадать с архитектурой, поддерживаемой кросс-инструментами).

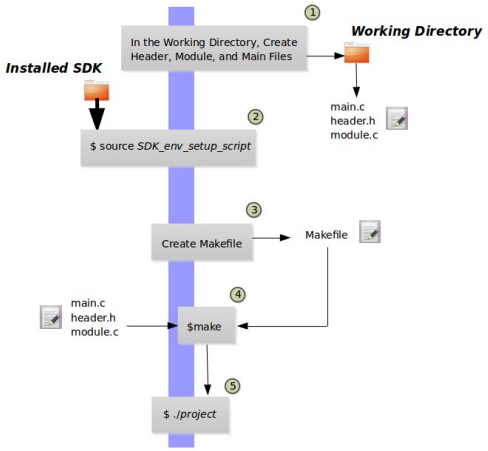
```
$ file ./tmp/usr/local/bin/hello
```

6. *Выполнение программы* на целевом оборудовании. Если целевым устройством является хост сборки, можно просто ввести команду ./tmp/usr/local/bin/hello, в результате чего на консоли появится сообщение "Hello World!".

4.2. Задания на основе Makefile

Простой проект на основе Makefile взаимодействует с переменными окружения, заданными сценарием организации среды кросс-разработки. Переменные подчиняются обычным правилам make.

Здесь представлен простой процесс разработки на основе Makefile с описанием использования переменных окружения и переменных Makefile.



Основной целью является рассмотрение 3 разных вариантов поведения переменных.

1. *Переменные из Makefile не отображаются на эквивалентные переменные, заданные сценарием организации среды SDK.* Поскольку совпадающие переменные не задаются в Makefile, они сохраняют значения, заданные установочным сценарием.
2. *Переменные из Makefile отображаются на эквивалентные переменные, заданные сценарием организации среды SDK.* В частности, установка переменных в Makefile в процессе сборки будет влиять на окружение путём переопределения переменных. Использоваться будут переменные из Makefile.
3. *Переменные из строки команды отображаются на эквивалентные переменные, заданные сценарием организации среды SDK.* Выполнение ведёт к переопределению переменных значениями из команды.

Независимо от способа установки переменных опция make -e отдаёт преимущество настройкам сценария организации среды SDK.

```
$ make -e target
```

В новой оболочке переменные для SDK не устанавливаются, пока не будет запущен сценарий организации среды, например, команда echo ${CC} выведет пустую (null) строку значения переменной для компилятора (CC).

Сценарий организации среды SDK для 64-битового хоста сборки и целевой архитектуры i586-tuned с образом core-image-sato, использующим выпуск YP 3.0, запускается приведённой ниже командой.

```
$ source /opt/poky/3.0/environment-setup-i586-poky-linux
$ echo ${CC}
i586-poky-linux-gcc -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux
```

Для иллюстрации снова рассмотрим простой пример Hello World!

1. *Создание и наполнение рабочего каталога*.

```
$ mkdir $HOME/helloworld
$ cd $HOME/helloworld
```

В рабочем каталоге нужно создать файл main.c, где вызывается функция, module.h с заголовками и module.c с определением функции, как показано ниже.

- main.c

```
#include "module.h"
void sample_func();
int main()
{
   sample_func();
   return 0;
}
```

- module.h

```
#include <stdio.h>
void sample_func();
```

- module.c

```
#include "module.h"
void sample_func()
```

```
{
        printf("Hello World!");
        printf("\n");
}
```

2. *Выполнение сценария организации среды кросс-разработки из каталога SDK.* Имя сценария начинается с environment-setup и включает архитектуру машины, а затем строку poky-linux. Например, сценарий из принятой по умолчанию установки SDK для 32-битовой архитектуры Intel x86 и выпуска Zeus YP запускается командой source /opt/poky/3.0/environment-setup-i586-poky-linux.

```
$ source /opt/poky/3.0/environment-setup-i586-poky-linux
```

3. *Создание Makefile.* В этом примере Makefile включает 2 строки, используемые для установки переменной CC. Одна строка идентична устанавливаемой сценарием организации среды SDK, а другая устанавливает в CC значение "gcc" (принятый по умолчанию компилятор GNU на хосте сборки).

```
# CC=i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux
# CC="gcc"
all: main.o module.o
    ${CC} main.o module.o -o target_bin
main.o: main.c module.h
        ${CC} -I . -c main.c
module.o: module.c module.h
        ${CC} -I . -c module.c
clean:
        rm -rf *.o
        rm target_bin
```

4. *Сборка проекта* с помощью команды make для создания выходного двоичного файла. Поскольку переменные в Makefile закомментированы, применяться будет значение CC, заданное сценарием организации среды SDK.

```
$ make
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc   -m32  -march=i586  --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux  main.o  module.o  -o
target_bin
```

Из приведённого вывода видно, что применялся компилятор, заданный значением CC в сценарии установки. Можно переопределить переменную CC, установив тоже значение путём удаления символа комментария из Makefile и повторного запуска make.

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile by uncommenting the line that sets CC to "gcc"
#
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

Здесь не использовался кросс-компилятор.

Следующий пример показывает переопределение переменной в команде (символы комментариев в Makefile восстановлены).

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile to comment out the line setting CC to "gcc"
#
$ make
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
i586-poky-linux-gcc   -m32   -march=i586   --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux   main.o   module.o   -o
target_bin
$ make clean
rm -rf *.o
rm target_bin
$ make CC="gcc"
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
```

Во втором случае команда задавала использование компилятора, переопределяющее установку SDK.

Последний пример использует установку в Makefile переменной "gcc", но применяется опция "-e" в команде make.

```
$ make clean
rm -rf *.o
rm target_bin
#
# Edit the Makefile to use "gcc"
#
$ make
gcc -I . -c main.c
gcc -I . -c module.c
gcc main.o module.o -o target_bin
$ make clean
rm -rf *.o
rm target_bin
$ make -e
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c main.c
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux -I . -c module.c
```

```
i586-poky-linux-gcc  -m32 -march=i586 --sysroot=/opt/poky/2.5/sysroots/i586-poky-linux main.o module.o -o target_bin
```

В результате указания опции -e применяются настройки SDK, независимо от Makefile.

5. *Выполнение программы.*

```
$ ./target_bin
Hello World!
```

При использовании кросс-компилятора для сборки target_bin под иную архитектуру, файл нужно запускать на соответствующем устройстве.

Приложение А. Получение SDK

А.1. Собранные установщики

Можно использовать имеющиеся собранные установщики, найдя и запустив соответствующий сценарий инсталляции SDK из состава YP. Таким способом можно выбрать и загрузить SDK для нужной архитектуры, а затем запустить сценарий для установки вручную, как описано ниже.

1. *Загрузка страницы установщиков* http://downloads.yoctoproject.org/releases/yocto/yocto-3.0/toolchain/
2. *Выбор каталога для хоста сборки* (например, i686 для 32-битовой машины или x86_64 для 64-битовой).
3. *Выбор и загрузка установщика SDK* с учётом хоста сборки, целевой системы и типа образа. Имена сценариев установки имеют форму poky-glibc-host_system-core-image-type-arch-toolchain[-ext]-release.sh, где host_system представляет хост разработки (i686 или x86_64), type указывает образ (sato или minimal), arch - архитектуру (aarch64, armv5e, core2-64, coretexa8hf-neon, i586, mips32r2, mips64 или ppc7400, а release - выпуск YP. Установщики для стандартных SDK не имеют суффикса -ext.

   Инструменты YP основаны на образе core-image-sato или core-image-minimal и включают двоичные файлы, подходящие для этих образов. Например, для 64-битового хоста сборки x86 и 64-битовой целевой системы core2 следует перейти в каталог x86_64 и загрузить файл (для расширенного SDK) poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-3.0.sh.
4. *Запуск установщика.* Например, для запуска из каталога Downloads в домашнем каталоге служит команда ~/Downloads/poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-3.0.sh. При выполнении сценария указывается каталог для инструментов. Структура каталогов показана в приложениях А.4. Структура каталогов стандартного SDK и А.5. Структура каталогов eSDK.

А.2. Сборка установщика SDK

Другим решением служит самостоятельная сборка установщика SDK, этапы которой описаны ниже.

1. *Организация среды сборки* BitBake и прочтите раздел Preparing the Build Host [1], где описана подготовка хоста сборки на машине Linux или машине с CROPS.
2. *Клонирование репозитория poky* в локальный каталог исходных кодов (локальный репозиторий poky) в соответствии с разделами Cloning the poky Repository, а также Checking Out by Branch in Poky и Checking Out by Tag in Poky [1], выбрав нужную ветвь.
3. *Инициализация среды сборки* путём запуска из корневого каталога дерева источников (poky) сценария организации среды oe-init-build-env для настройки окружения системы сборки OE на сборочном хосте.

```
$ source oe-init-build-env
```

   Этот сценарий создаст каталог сборки (по умолчанию build) внутри каталога источников, а также сделает этот каталог текущим.
4. *Проверка корректности выбора машины.* Переменная MACHINE в файле local.conf внутри каталога сборки должна соответствовать архитектуре, для которой будет выполняться сборка.
5. *Проверка корректности выбора машины для SDK.* При сборке инструментов для архитектуры, отличающейся от архитектуры хоста сборки следует убедиться, что переменная SDKMACHINE в файле local.conf сборочного каталога установлена корректно. При сборке установщика eSDK переменная SDKMACHINE должна указывать архитектуру машины, применяемой для сборки установщика. Некорректная установка SDKMACHINE ведёт к отказу сборки с выводом сообщения, подобного приведённому ниже.

   Расширяемый SDK в данном случае можно собрать лишь для той архитектуры, к которой относится хост сборки. Для переменной SDK_ARCH установлено значение i686 (вероятно через SDKMACHINE), которое отличается от архитектуры сборочного хоста (x86_64). Продолжение невозможно.
6. *Сборка установщика SDK.* Для сборки и создания образа стандартного SDK служит команда вида bitbake image -c populate_sdk, для расширенного - bitbake image -c populate_sdk_ext (параметр image должен указывать используемый образ, например, core-image-sato). По завершении работы bitbake установщик будет находиться в каталоге tmp/deploy/sdk внутри каталога сборки.

   По умолчанию приведённые выше команды BitBake не создают статических двоичных файлов. Если планируется использование инструментов для сборки статических библиотек, нужно убедиться в наличии в SDK соответствующих библиотек разработки. Переменная TOOLCHAIN_TARGET_TASK в файле local.conf должна быть установлена до сборки установщика SDK и иметь вид TOOLCHAIN_TARGET_TASK_append = " libc-staticdev".
7. *Запуск установщика* из каталога tmp/deploy/sdk в сборочном каталоге. Например,

```
$ cd ~/poky/build/tmp/deploy/sdk
$ ./poky-glibc-x86_64-core-image-sato-core2-64-toolchain-ext-3.0.sh
```

При выполнении сценария нужно будет указать каталог установки. Структура каталогов описана в приложениях А.4. Структура каталогов стандартного SDK и А.5. Структура каталогов eSDK.

А.3. Извлечение корневой файловой системы

После установки инструментов в некоторых случаях может потребоваться извлечение корневой файловой системы:

- загрузка образа с использованием NFS;
- использование корневой файловой системы в качестве целевой sysroot;
- разработка целевых приложений, использующих корневую файловую систему в качестве целевой sysroot.

Этапы извлечения корневой файловой системы описаны ниже.

1. *Нахождение и загрузка архива подготовленного образа корневой файловой системы. Файлы таких образов доступны на странице* Index of Releases в каталоге machines. Образы представлены в архивах *.tar.bz2 для поддерживаемых машин, а также образы *.ext4 для использования с QEMU. Имена файлов с образами имеют форму core-image-profile-arch.tar.bz2, где profile указывает профиль образа файловой системы (lsb, lsb-dev, lsb-sdk, minimal, minimal-dev, minimal-initramfs, sato, sato-dev, sato-sdk, sato-sdk-ptest). Описания образов приведены в разделе Images. Строка arch в имени файла указывает целевую архитектуру (beaglebone-yocto, beaglebone-yocto-lsb, edgerouter, edgerouter-lsb, genericx86, genericx86-64, genericx86-64-lsb, genericx86-lsb, mpc8315e-rdb, mpc8315e-rdb-lsb, qemu*). Корневые файловые системы в составе YP основаны на образах core-image-sato и core-image-minimal.

   Например, для целевого устройства BeagleBone и образа core-image-sato-sdk следует выбрать файл core-image-sato-sdk-beaglebone-yocto.tar.bz2.

2. *Инициализация среды кросс-разработки* с помощью команды source для сценария настройки среды. Сценарий размещается в каталоге установки инструментов (например, poky_sdk). Ниже приведен пример запуска сценария для установки, описанной в приложении А.1. Собранные установщики

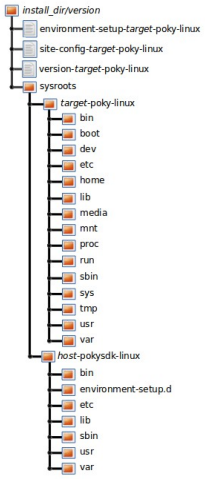
   ```
   $ source ~/poky_sdk/environment-setup-core2-64-poky-linux
   ```

3. *Извлечение корневой файловой системы* с помощью команды runqemu-extract-sdk. Например, для извлечения корневой файловой системы из образа, загруженного со страницы Index of Releases, в каталог core2-64-sato служит команда runqemu-extract-sdk ~/Downloads/core-image-sato-sdk-beaglebone-yocto.tar.bz2 ~/beaglebone-sato. После этого можно указать beablebone-sato в качестве целевой sysroot.

А.4. Структура каталогов стандартного SDK

На рисунке показана структура каталогов стандартного SDK, создаваемая установочным сценарием *.sh. Установленный SDK включает сценарий настройки среды, файл конфигурации для цели и корневую файловую систему для разработки объектов, предназначенных для целевой платформы.

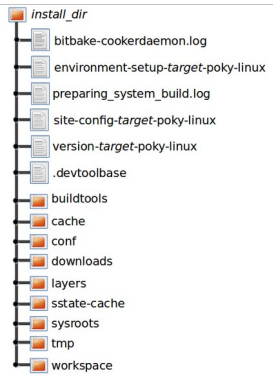
Наклонный шрифт на рисунке показывает заменяемые части имён файлов и каталогов. Например, *install_dir/version* является каталогом установки SDK, который по умолчанию называется /opt/poky/, а *version* представляет конкретный выпуск SDK (например, 3.0). Кроме того, target представляет целевую архитектуру (например, i586), а host - архитектуру системы разработки (например, x86_64). Таким образом, полные имена каталогов в sysroots могут быть i586-poky-linux (цель) и x86_64-pokysdk-linux (хост).



А.5. Структура каталогов eSDK

На рисунке показана структура каталогов eSDK, создаваемая установочным сценарием *.sh.

Структура каталогов eSDK отличается от структуры стандартного SDK отсутствием отдельных частей для хоста и цели, поскольку здесь они не разделяются. Расширяемый SDK содержит встроенную копию системы сборки OE со своими sysroot.



В структуре каталогов следует отметить сценарий организации среды для SDK, файл конфигурации для цели и журнальные файлы (log) для сценария подготовки системы сборки OE, запускаемого установщиком и BitBake.

Наклонный шрифт на рисунке показывает заменяемые части имён файлов и каталогов. Например, *install_dir/version* является каталогом установки SDK, который по умолчанию называется poky_sdk, а target представляет целевую архитектуру (например, i586).

Приложение B. Настройка eSDK

B.1. Конфигурация eSDK

Основной частью расширяемого SDK является настроенная копия системы сборки OE, из которой пакет был создан. Конфигурация SDK выводится с использованием системы сборки и приведённых ниже фильтров, которые система сборки OE применяет к файлам local.conf и auto.conf.

- Переменные, начинающиеся с /, исключаются в предположении, что это пути, зависящие от хоста сборки.
- Переменные из SDK_LOCAL_CONF_BLACKLIST исключаются, поскольку их не разрешено передавать из системы сборки OE в конфигурацию eSDK. Обычно эти переменные специфичны для машины, где работает система сборки и могут вызывать проблемы в конфигурации eSDK. Список переменных, исключаемых по умолчанию, приведен в описании переменной SDK_LOCAL_CONF_BLACKLIST [2].
- Переменные из SDK_LOCAL_CONF_WHITELIST включаются и могут переопределять предшествующие фильтры. По умолчанию список пуст.
- Глобально наследуемые классы с INHERIT из переменной SDK_INHERIT_BLACKLIST отключаются. Это стандартный метод исключения классов, которые могут создавать проблемы или не нужны в контексте SDK. По умолчанию список включает классы buildhistory и icecc.

Кроме того, при наличии файла conf/sdk-extra.conf его содержимое добавляется в конце conf/local.conf создаваемого SDK без фильтрации. Файл sdk-extra.conf полезен в тех случаях, когда нужно установить переменную для SDK, но не для системы сборки OE, используемой для создания SDK.

B.2. Настройка eSDK для хоста сборки

В большинстве случаев принятым по умолчанию для eSDK значениям следует работать с настройками хоста сборки. Однако в некоторых случаях могут потребоваться дополнительные настройки, описанные ниже.

- Если конфигурация SDK наследует дополнительные классы через переменную INHERIT и эти классы не следует включать в SDK, их можно отключить через переменную SDK_INHERIT_BLACKLIST, как описано в предыдущем параграфе. По умолчанию SDK_INHERIT_BLACKLIST устанавливается с помощью оператора ?=, поэтому нужно определить весь список с использованием = или добавлять значения с помощью _append или оператора +=. Описания операторов приведены в разделе Basic Syntax [3].
- При наличии классов или заданий, добавляющих задачи в стандартный поток сборки (задачи, выполняемые как сборка заданий, а не вызываемые явно), нужно выполнить одно из указанных действий:
    - после проверки принадлежности задачи к общему состоянию (т. е. их вывод сохраняется и может быть восстановлен из общего состояния) или её способности быстро создаваться из задачи с общим состоянием имя задачи добавляется в SDK_RECRDEP_TASKS;
    - отключаются все задачи, добавленные классом, если функциональность класса не нужна в eSDK; для отключения задач класс указывается в переменной SDK_INHERIT_BLACKLIST, как описано выше.
- Как правило настраивается зеркало общего состояния, чтобы пользователи SDK могли добавлять элементы после установки SDK без сборки их из исходного кода (B.6. Дополнительные компоненты eSDK).
- Для упрощения процедуры обновления SDK следует установить переменную SDK_UPDATE_URL (B.4. Обновление eSDK после установки.
- При корректировке списка файлов и каталогов в переменной COREBASE (сверх уровней, добавленных в bblayers.conf) нужно указать эти файлы в переменной COREBASE_FILES для их копирования в SDK.
- При использовании системой сборки OE своего сценария настройки окружения (не oe-init-build-env) нужно указать этот сценарий в переменной OE_INIT_ENV_SCRIPT. Это изменение нужно также отразить в переменной COREBASE_FILES, как описано выше.

B.3. Изменение названия установщика eSDK

Можно установить отображаемое имя установщика SDK с помощью переменной SDK_TITLE и пересборки установщика (A.2. Сборка установщика SDK). По умолчанию заголовок берётся из DISTRO_NAME, а при отсутствии этой переменной - из DISTRO. Класс populate_sdk_base определяет принятое по умолчанию значение в форме SDK_TITLE ??= "${@d.getVar('DISTRO_NAME') or d.getVar('DISTRO')} SDK".

Существуют разные способы изменения этой переменной, но лучше установить её в файле конфигурации дистрибутива. Это задаёт имя установщика SDK в рамках дистрибутива. Предположим, например, наличие уровня meta-mydistro и использование в нем иерархии файлов как в дистрибутиве poky. В этом случае можно обновить переменную SDK_TITLE в файле ~/meta-mydistro/conf/distro/mydistro.conf, используя форму SDK_TITLE = "your_title".

B.4. Обновление eSDK после установки

При изменении конфигурации или метаданных для учёта этих изменений в установленном SDK нужно выполнить дополнительные действия. Это позволит любому пользователю установленного SDK обновить пакет с помощью команды devtool sdk-update.

1. Создаётся каталог, доступный по протоколу HTTP или HTTPS (например, с помощью сервера Apache или Nginx) и содержащий опубликованный SDK.
2. Устанавливается переменная SDK_UPDATE_URL с указанием соответствующего HTTP или HTTPS URL. Установка переменной ведёт к тому, что любой SDK собранный по умолчанию с этим URL, будет передавать идентификатор команде devtool sdk-update, как описано в разделе 2.9. Обновление установленного eSDK.
3. Обычным способом собирается eSDK (bitbake -c populate_sdk_ext imagename).
4. SDK публикуется с помощью команды oe-publish-sdk some_path/sdk-installer.sh path_to_shared_http_directory. Этот этап нужно повторять при каждом изменении SDK, которое следует делать доступным.

Выполнение этих действий позволит обновлять установленные SDK с помощью команды devtool sdk-update для получения и применения изменений, как описано в разделе 2.9. Обновление установленного eSDK.

B.5. Изменение каталога установки eSDK

При сборке установщика eSDK каталог установки по умолчанию определяется переменными DISTRO и SDKEXTPATH из класса populate_sdk_base в форме SDKEXTPATH ??= "~/${@d.getVar('DISTRO')}_sdk". Этот каталог можно изменить через переменную SDKEXTPATH.

Существуют разные способы изменения этой переменной, но лучше установить её в файле конфигурации дистрибутива. Это задаёт имя установщика SDK в рамках дистрибутива. Предположим, например, наличие уровня meta-mydistro и использование в нем иерархии файлов как в дистрибутиве poky. В этом случае можно обновить переменную SDKEXTPATH в файле ~/meta-mydistro/conf/distro/mydistro.conf, используя форму SDKEXTPATH = "some_path_for_your_installed_sdk".

При запуске собранного установщика пользователю предлагается принять каталог some_path_for_your_installed_sdk для установки eSDK или указать свой каталог.

B.6. Дополнительные компоненты eSDK

Чтобы пользователи собранного eSDK могли добавлять элементы без их сборки из исходного кода, нужно выполнить ряд действий.

1. Проверка наличия требуемых для установки элементов:
   - явная сборка элементов с использованием одного или нескольких метазаданий;
   - сборка цели world с установкой EXCLUDE_FROM_WORLD_pn-recipename для ненужных заданий (см. описание EXCLUDE_FROM_WORLD).
2. Раскрытие каталога sstate-cache, создаваемого сборкой (обычно через сервер Apache или Nginx).
3. Установка конфигурации, позволяющей созданному SDK узнать свои настройки, для чего нужно задать переменную SSTATE_MIRRORS = "file://.* http://example.com/some_path/sstate-cache/PATH".
   - Если указанное зеркало подходит как для системы сборки OE, использованной для создания SDK, так и для самого SDK (т. е. зеркало доступно обоим или при быстром выходе из строя на стороне системы сборки OE содержимое не будет препятствовать сборке), можно установить переменную в файле local.conf или файле конфигурации дистрибутива. Затем переменную можно включить в «белый список» SDK в форме SDK_LOCAL_CONF_WHITELIST = "SSTATE_MIRRORS".
   - Дополнительно можно установить переменную SSTATE_MIRRORS только для SDK, создав файл conf/sdk-extra.conf в каталоге сборки или на любом уровне и поместив туда SSTATE_MIRRORS.

   Второй вариант является наиболее безопасным для установки SSTATE_MIRRORS.

B.7. Минимизация размера установщика eSDK

По умолчанию eSDK включает элементы общего состояния для всего, что нужно при восстановлении образа, для которого собран SDK. Такая сборка ведёт к размеру SDK, превышающему 1 Гбайт. Если это вызывает проблемы, можно собрать SDK, которого достаточно для установки и есть доступ к команде devtool, путём задания SDK_EXT_TYPE = "minimal". Это приведёт к снижению размера SDK примерно до 35 Мбайт, что существенно ускорит загрузку и установку. Однако следует помнить, что этот вариант не установит библиотеки и инструменты, которые придётся устанавливать «на лету» с помощью devtool или явной команды devtool sdk-install.

В большинстве случаев при создании минимального SDK нужно включать также информацию о спектре пакетов, создаваемых системой. Это особенно важно, поскольку команда devtool add может эффективно сопоставлять зависимости в дереве источников с заданиями. Кроме того, это позволяет команде devtool search возвращать полезные результаты. Для упрощения задачи следует установить SDK_INCLUDE_PKGDATA = "1" (см. SDK_INCLUDE_PKGDATA).

Установка переменной SDK_INCLUDE_PKGDATA ведёт к тому, что цель world собирается так, чтобы предоставить информацию обо всех доступных заданиях. Это существенно увеличивает время сборки и повышает размер SDK на 30-80 Мбайт в зависимости от числа включённых в конфигурацию заданий.

Можно использовать переопределение EXCLUDE_FROM_WORLD_pn-recipename для исключения ненужных заданий, однако считается, что нужно собрать цель world для предоставления дополнительных элементов SDK. Поэтому сборка world не должна вызывать неприемлемых издержек в большинстве случаев.

При установке SDK_EXT_TYPE = "minimal" предоставление зеркала общего состояния является обязательным, чтобы элементы можно было устанавливать по мере необходимости (см. B.6. Дополнительные компоненты eSDK).

Можно явно управлять включением инструментов при сборке SDK через установку SDK_INCLUDE_TOOLCHAIN = "1". в частности, полезно включать инструменты при установке SDK_EXT_TYPE = "minimal", где они исключены по умолчанию. Полезно также включать инструменты при создании небольших SDK для использования с IDE или иными средствами, где нужно избежать дополнительных шагов по установке инструментария.

Приложение C. Настройка стандартного SDK

C.1. Добавление пакетов

При сборке стандартного SDK по команде bitbake -c populate_sdk в SDK включается принятый по умолчанию набор пакетов, которым управляют переменные TOOLCHAIN_HOST_TASK и TOOLCHAIN_TARGET_TASK. Для включения дополнительных пакетов в инструментарий хоста служит переменная TOOLCHAIN_HOST_TASK, а для целевой платформы - TOOLCHAIN_TARGET_TASK.

C.2. Добавление документации API

Можно включить документацию API, а также другую документацию из заданий в стандартном SDK путём добавления api-documentation в переменную DISTRO_FEATURES, как показано ниже.

```
DISTRO_FEATURES_append = " api-documentation"
```

Установка переменной заставит систему сборки OE подготовить документацию и включит её в стандартный SDK.

Литература

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

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

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

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru