Руководство для разработчиков Yocto Project Linux Kernel

Scott Rifenbark, <srifenbark@gmail.com>

Scotty's Documentation Services, INC

Оглавление

Глава 1. Введение

1.1. Обзор

Независимо от целей использования Yocto Project (YP), велика вероятность работы с ядром Linux. В этом документе описано как организовать сборочный хост для поддержки работы с ядром, приведена вводная информация о процессах разработки ядра, а также базовые сведения о метаданных ядра (Metadata), описывающих задачи, доступные с помощью инструментов ядра, и применение Metadata для работы с ядром в YP, а также рассмотрены вопросы разработки командой поддержки YP репозиториев Git и Metadata для ядра.

Каждый выпуск YP имеет набор заданий (recipe) Yocto Linux kernel, Git-репозитории которых можно увидеть в Yocto Source Repositories [3] под заголовком Yocto Linux Kernel. Новые задания для выпусков отслеживают разработки ядра Linux с сайта http://www.kernel.org и добавляют поддержку новых платформ. Прежние задания в выпуске обновляются и поддерживаются по меньшей мере до следующего выпуска YP. По мере внесения изменений эти выпуски обновляются и включают все новое из проекта LTSI[1]. Дополнительная информация о ядрах Yocto Linux и LTSI приведена в приложении A.1. Разработка и поддержка YP Kernel.

В проект включено задание для работы с ядром linux-yocto-dev.bb, которое позволяет ознакомится с находящимися в разработке ядром Yocto Linux и Metadata.

YP также включает мощный набор инструментов ядра для управления исходными кодами Yocto Linux и параметрами конфигурации. Эти инструменты могут служить для внесения единичных изменений, применения множества исправлений (patch) или разработки своих компонент ядра.

В частности, эти инструменты позволяют генерировать фрагменты конфигурации, которые задают лишь то, что нужно сделать, не затрагивая остального. Во фрагменты конфигурации требуется включать лишь видимые опции верхнего уровня, представляемые системой Yocto Linux menuconfig. В отличие от этого полный файл Yocto Linux .config включает все автоматически выбираемые опции CONFIG. Это снижает объем работы по поддержке и позволяет дополнительно разделить конфигурацию удобным для вашего проекта способом. Обычно разделяются компоненты, связанные с оборудованием и правилами. Например, ваше ядро может поддерживать файловые системы proc и sys, а поддержка звука, USB и некоторых драйверов нужна лишь для отдельных плат. Задавая эти параметры отдельно, вы потом можете собрать их воедино, но поддерживать каждый фрагмент независимо. Аналогичная логика применима и для разделения вносимых в код изменений.

Если вы не поддерживаете своего дерева исходных кодов ядра и нужны лишь незначительные изменения в исходных файлах, выпущенные задания обеспечивают надёжный способ наложить ваши правки. Это позволяет сохранить интеграцию ядра с тестированием производительности в процессе работы с YP.

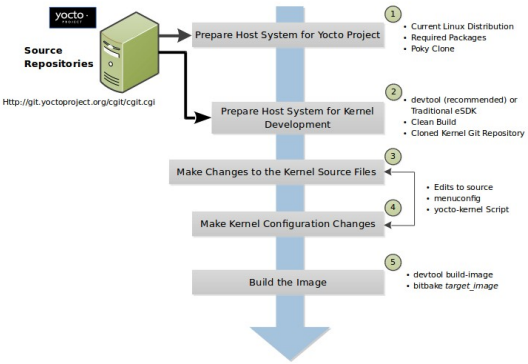
Если же у вас имеется своё дерево исходных кодов ядра Linux и вы не можете совместить его ни с одним из официальных заданий Yocto Linux kernel, можно использовать инструменты ядра YP Linux с вашими исходными кодами.

Далее приведены инструкции по решению конкретных задач при работе с ядром Linux. Эти инструкции предполагают знакомство читателя с заданиями BitBake и основными инструментами для разработки систем с открытым кодом. Эти знания облегчат понимание процессов работы с заданиями для ядра (kernel recipe). Дополнительную информацию можно найти по ссылкам, приведённым в конце документа.

1.2. Рабочий процесс изменения ядра

Изменение ядра включает изменения в YP, которые могут включать смену параметров конфигурации, а также добавление в ядро новых заданий. Конфигурационные изменения могут быть добавлены в форме фрагментов конфигурации, а изменения заданий будут проходить через область recipes-kernel на созданном вами уровне.

В этом параграфе представлен общий обзор процесса изменения ядра в YP. На рисунке и в следующем за ним списке представлена базовая информация и ссылки на дополнительные источники.



1. *Организация на хосте среды разработки с использованием YP* - в разделе Setting Up the Development Host to Use the Yocto Project [4] описаны варианты организации сборочного хоста YP.

2. *Организация системы разработки для работы с ядром.* Рекомендуется использовать devtool и расширяемый SDK[2] (eSDK). Другим вариантом является использование традиционных методов разработки ядра в среде YP. Для обоих вариантов приведены этапы организации среды разработки.

   Для использования devtool и eSDK нужна «чистая» сборка образа и установка подходящего eSDK (см. параграф 2.1.1. Подготовка к работе с использованием devtool).

[1]Long Term Support Initiative - инициатива долгосрочной поддержки.
[2]Software Development Kit - набор (комплект) для разработки программ.

Для использования традиционных методов нужны исходные коды ядра в виде изолированного от среды разработки локального репозитория Git (2.1.2. Подготовка к традиционной работе с ядром).

3. *Внесение изменений в исходный код ядра (при необходимости).* Изменение ядра не всегда требует вмешательства в файлы исходного кода. Однако при необходимости можно внести изменить файлы каталога сборки eSDK, если используется devtool (2.4. Использование devtool для применения изменений к ядру).

   При использовании традиционных методов редактируются файлы в локальном репозитории Git (см. параграф 2.5. Традиционные методы внесения изменений в ядро).

4. *Изменение конфигурации ядра (при необходимости).* Если нужно изменить конфигурацию ядра, можно воспользоваться командой menuconfig (2.6.1. Использование menuconfig), которая позволяет в интерактивном режиме изменить и проверить конфигурацию ядра. При сохранении результатов menuconfig обновляет файл .config.

   Старайтесь не редактировать файл .config в каталоге сборки напрямую, поскольку это может приводить к непредсказуемым результатам, когда система сборки OpenEmbedded (OE) регенерирует конфигурационный файл.

   После создания нужной конфигурации с помощью menuconfig и сохранения файла можно напрямую сравнить полученный файл .config с имеющимся вариантом конфигурации и собрать изменения в файл фрагмента конфигурации (2.6.3. Создание фрагментов конфигурации), указываемый в файле .bbappend для ядра.

   При работе с уровнем BSP[1] и необходимости изменить конфигурацию ядра в BSP также можно использовать menuconfig.

5. *Сборка ядра с учётом изменений.* Процесс сборки зависит от целевой платформы (реальная плата или QEMU).

Далее рассматриваются типовые процедуры работы с ядром, использование метаданных и концепции поддержки ядра Yocto Linux.

Глава 2. Базовые задачи

В этой главе описано несколько базовых задач, которые выполняются при работе с ядрами YP Linux. Эти задачи включают подготовку хоста для работы, подготовку уровня (layer), изменение имеющихся заданий, наложение «заплаток» (patch) для ядра, настройку конфигурации ядра, работу с вашими файлами исходного кода и встраивание внешних модулей.

Представленные здесь примеры подходят для YP 2.4 и последующих выпусков.

2.1. Подготовка сборочного хоста для работы с ядром

Перед началом работы с ядром нужно подготовить ваш хост для использования YP (см. раздел Preparing the Build Host [4]). Частью подготовки системы является создание локального репозитория Source Directory (poky) [1] на хосте сборки. Этот процесс подробно описан в разделе Cloning the poky Repository [4].

Выберите подходящую ветвь разработки или создайте свою локальную ветвь, указав определённый тег для получения нужной версии YP. Дополнительная информация по этим вопросам представлена в разделах Checking Out by Branch in Poky и Checking Out by Tag in Poky [4].

Для работы с ядрами лучше всего подходит инструмент (команда) devtool, а не традиционные методы работы с ядрами. В последующих параграфах рассматриваются оба подхода.

2.1.1. Подготовка к работе с использованием devtool

Ниже описаны подготовительные операции для обновления образа ядра с помощью devtool. По завершении процедуры вы получите «чистый» образ ядра и будете готовы к внесению изменений, описанных в параграфе 2.4. Использование devtool для применения изменений к ядру.

1. *Инициализация среды BitBake.* Перед созданием расширяемого SDK нужно инициализировать среду сборки BitBake с помощью соответствующего сценария (например, oe-init-build-env).

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

   Приведённые команды предполагают, что репозитории исходных кодов (poky) были клонированы с помощью Git в локальный каталог poky в домашнем каталоге пользователя.

2. *Подготовка файла local.conf.* По умолчанию для переменной MACHINE установлено значение qemux86, предполагающее сборку для эмулятора QEMU в 32-битовом режиме. Если нужен другой вариант, следует указать корректное значение переменной MACHINE в файле conf/local.conf file локального каталога сборки (~/poky/build в этом примере).

   Нужно также установить значение переменной MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS для включения модулей ядра. В примере используется принятое по умолчанию значение qemux86 переменной MACHINE, но нужно добавить в файл строку

   ```
   MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-modules"
   ```

3. *Создание уровня для исправлений (patch).* Нужно создать уровень для размещения «заплаток» к образу ядра. Для этого служит команда bitbake-layers create-layer, как показано ниже.

   ```
   $ cd ~/poky/build
   $ bitbake-layers create-layer ../../meta-mylayer
   NOTE: Starting bitbake server...
   ```

[1]Board Support Package - пакет поддержки плат.

```
Add your new layer with 'bitbake-layers add-layer ../../meta-mylayer'
$
```

Базовые сведения о работе с общими уровнями и уровнями BSP приведены в разделе Understanding and Creating Layers [4] и разделе BSP Layers [5]. Использование команды bitbake-layers create-layer для быстрого создания уровня описано в разделе Creating a General Layer Using the bitbake-layers Script [4].

4. *Добавление в среду разработки BitBake информации о новом уровне.* При создании уровня нужно добавить его в переменную BBLAYERS файла bblayers.conf, как показано ниже.

```
$ cd ~/poky/build
$ bitbake-layers add-layer ../../meta-mylayer
NOTE: Starting bitbake server...
$
```

5. *Сборка расширяемого SDK.* С помощью BitBake создаётся расширяемый SDK для работы с образами, предназначенными для QEMU.

```
$ cd ~/poky/build
$ bitbake core-image-minimal -c populate_sdk_ext
```

После завершения сборки файл установки SDK (.sh) будет размещаться в каталоге ~/poky/build/tmp/deploy/sdk. В этом примере файл установки называется poky-glibc-x86_64-core-image-minimal-i586-toolchain-ext-2.7.1.sh.

6. *Установка ESDK.* Используйте указанную выше команду для установки SDK. Например, для установки в принятом по умолчанию каталоге ~/poky_sdk следует ввести команды, показанные ниже.

```
$ cd ~/poky/build/tmp/deploy/sdk
$ ./poky-glibc-x86_64-core-image-minimal-i586-toolchain-ext-2.7.1.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 2.7.1
==========================================================================
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
Initializing tasks: 100% |############### ##########################################################| Time: 0:00:04
Checking sstate mirror object availability: 100% |#######################################| Time: 0:00:00
Parsing recipes: 100% |#################################################################| Time: 0:00:33
Initializing tasks: 100% |#################################################################| Time: 0:00:00
done
```

Это говорит об успешной установке и готовности SDK к работе. Каждый раз перед началом использования SDK в новой сессии нужно активировать сценарий настройки окружения, например

```
$ . /home/scottrif/poky_sdk/environment-setup-i586-poky-linux
```

7. *Новая терминальная сессия для работы с ESDK.* Для SDK нужно организовать новый терминал. Нельзя применять командный процессор BitBake для сборки установщика. После открытия новой терминальной сессии нужно запустить сценарий настройки среды SDK, как было указано при завершении установки SDK.

```
$ source ~/poky_sdk/environment-setup-i586-poky-linux
"SDK environment now set up; additionally you may now run devtool to perform development tasks.
Run devtool --help for further details.
```

Вывод предупреждения о попытке использовать общую среду для SDK и BitBake говорит о том, что вы не организовали другую сессию.

8. *Сборка чистого образа.* Финальным этапом подготовки к работе с ядром является создание исходного образа с использованием devtool в новой терминальной сессии, где установлен и инициализирован пакет SDK.

```
$ devtool build-image
Parsing recipes: 100% |##########################################| Time: 0:00:05
Parsing of 830 .bb files complete (0 cached, 830 parsed). 1299 targets, 47 skipped, 0 masked, 0 errors.
WARNING: No packages to add, building image core-image-minimal unmodified
Loading cache: 100% |############################################| Time: 0:00:00
Loaded 1299 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies
Initializing tasks: 100% |#######################################| Time: 0:00:07
Checking sstate mirror object availability: 100% |################| Time: 0:00:00
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2866 tasks of which 2604 didn't need to be rerun and all succeeded.
NOTE: Successfully built core-image-minimal. You can find output files in
/home/scottrif/poky_sdk/tmp/deploy/images/qemux86
```

Если сборка выполняется для реального оборудования, а не эмулятора, вам потребуется переписать образ на карту памяти, которая будет устанавливаться в устройство. Пример для платы Minnowboard приведен на странице TipsAndTricks/KernelDevelopmentWithEsdk.

С этого момента можно начать работу ядром в SDK. Наложение «заплаток» на ядро описано в разделе 2.4. Использование devtool для применения изменений к ядру.

2.1.2. Подготовка к традиционной работе с ядром

Подготовка к использованию традиционных методов работы с ядром в YP включает большинство описанных в предыдущем параграфе шагов. Однако нужно организовать локальную копию дерева исходных кодов ядра для редактирования этих файлов.

Выполните перечисленные ниже операции для обновления образа ядра с использованием традиционных методов в YP. По завершении процедуры вы будете готовы вносить изменения в ядро, как описано в параграфе 2.5. Традиционные методы внесения изменений в ядро.

1. *Инициализация среды BitBake.* Перед началом работы нужно инициализировать среду сборки BitBake с помощью соответствующего сценария (например, oe-init-build-env). В этом примере также нужно убедиться что в качестве локальной ветви для poky выбрана Yocto Project Warrior. Если это не так, перейдите (checkout) на нужную ветвь, как описано в параграфе Checking out by Branch in Poky [4].

```
$ cd ~/poky
$ git branch master
* Warrior
$ source oe-init-build-env
```

   В приведённых выше командах предполагается, что каталог Source Repositories (poky) был клонирован с помощью Git в локальный репозиторий с именем poky.

2. *Подготовка файла* local.conf. По умолчанию для переменной MACHINE установлено значение qemux86, предполагающее сборку для эмулятора QEMU в 32-битовом режиме. Если нужен другой вариант, следует указать корректное значение переменной MACHINE в файле conf/local.conf локального каталога сборки (~/poky/build в этом примере).

   Нужно также установить значение переменной MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS для включения модулей ядра. В примере используется принятое по умолчанию значение qemux86 переменной MACHINE, но нужно добавить в файл строку

```
MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS += "kernel-modules"
```

3. *Создание уровня для исправлений (patch).* Нужно создать уровень для размещения «заплаток» к ядру. Для этого служит команда bitbake-layers create-layer, как показано ниже.

```
$ cd ~/poky/build
$ bitbake-layers create-layer ../../meta-mylayer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer ../../meta-mylayer'
$
```

   Базовые сведения о работе с общими уровнями и уровнями BSP приведены в разделе Understanding and Creating Layers [4] и разделе BSP Layers [5]. Использование команды bitbake-layers create-layer для быстрого создания уровня описано в разделе Creating a General Layer Using the bitbake-layers Script [4].

4. *Информирование среды разработки BitBake о новом уровне.* Нужно добавить переменную BBLAYERS в файл bblayers.conf, как показано ниже.

```
$ cd ~/poky/build
$ bitbake-layers add-layer ../../meta-mylayer
NOTE: Starting bitbake server ...
$
```

5. *Создание локальной копии репозитория Kernel Git.* Вы можете найти репозитории Git для поддерживаемых в YP ядер в разделе Yocto Linux Kernel по ссылке http://git.yoctoproject.org.

   Для простоты рекомендуется создавать копию репозитория ядра за пределами каталога исходных кодов, который обычно именуется poky. Убедитесь также в выборе ветви standard/base. Ниже приведены команды для создания локальной копии ядра linux-yocto-4.12 из ветви standard/base. Ядро linux-yocto-4.12 можно использовать с YP версии 2.4 и выше.

```
$ cd ~
$ git clone git://git.yoctoproject.org/linux-yocto-4.12 --branch standard/base
Cloning into 'linux-yocto-4.12'...
remote: Counting objects: 6097195, done.
remote: Compressing objects: 100% (901026/901026), done.
remote: Total 6097195 (delta 5152604), reused 6096847 (delta 5152256)
Receiving objects: 100% (6097195/6097195), 1.24 GiB | 7.81 MiB/s, done.
Resolving deltas: 100% (5152604/5152604), done.
Checking connectivity... done.
Checking out files: 100% (59846/59846), done.
```

6. *Создание локальной копии репозитория Kernel Cache Git.* Для простоты рекомендуется создавать копию репозитория ядра за пределами каталога исходных кодов, который обычно именуется poky. Убедитесь также в выборе ветви yocto-4.12. Ниже приведены команды для создания локальной копии yocto-kernel-cache (ветвь yocto-4.12).

```
$ cd ~
$ git clone git://git.yoctoproject.org/yocto-kernel-cache --branch yocto-4.12
Cloning into 'yocto-kernel-cache'...
remote: Counting objects: 22639, done.
remote: Compressing objects: 100% (9761/9761), done.
remote: Total 22639 (delta 12400), reused 22586 (delta 12347)
Receiving objects: 100% (22639/22639), 22.34 MiB | 6.27 MiB/s, done.
Resolving deltas: 100% (12400/12400), done.
Checking connectivity... done.
```

С этого момента можно начать работу с ядром с помощью традиционных средств. Наложение «заплаток» на ядро описано в разделе 2.5. Традиционные методы внесения изменений в ядро.

2.2. Создание и подготовка уровня

Если вы планируете менять задания для ядра, рекомендуется создать и подготовить свой уровень, с которым вы будете работать. Этот уровень будет включать свой файл дополнения (.bbappend) и обеспечивать удобный механизм

для создания файлов заданий (.bb), а также для хранения и применения «заплаток» к ядру. Работа с уровнями описана в разделе Understanding and Creating Layers [4].

YP включает множество инструментов для упрощения работы. Одним из таких инструментов является команда bitbake-layers create-layer (см. Creating a General Layer Using the bitbake-layers Script [4]).

Для лучшего понимания роли создаваемого уровня при работе с ядром далее описано создание уровня без использования этих инструментов. Описанные этапы предполагают создание уровня mylayer в домашнем каталоге.

1. *Создание структуры уровня*

```
$ cd $HOME
$ mkdir meta-mylayer
$ mkdir meta-mylayer/conf
$ mkdir meta-mylayer/recipes-kernel
$ mkdir meta-mylayer/recipes-kernel/linux
$ mkdir meta-mylayer/recipes-kernel/linux/linux-yocto
```

   Каталог содержит конфигурационные файлы, а recipes-kernel - файл дополнения и файлы исправлений (patch).

2. *Создание файла конфигурации уровня.* Перейдите в каталог meta-mylayer/conf и создайте файл layer.conf, как показано ниже.

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

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

BBFILE_COLLECTIONS += "mylayer"
BBFILE_PATTERN_mylayer = "^${LAYERDIR}/"
BBFILE_PRIORITY_mylayer = "5"
```

   Обратите внимание на mylayer в трёх последних строках.

3. *Создание файла добавления заданий.* Перейдите в каталог meta-mylayer/recipes-kernel/linux и создайте файл добавления для ядра. В приведённом примере используется ядро linux-yocto-4.12. Поэтому файл добавления будет иметь имя linux-yocto_4.12.bbappend:

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

SRC_URI_append += "file://patch-file-one"
SRC_URI_append += "file://patch-file-two"
SRC_URI_append += "file://patch-file-three"
```

   Операторы FILESEXTRAPATHS и SRC_URI позволяют системе сборки OE найти файлы добавления. Дополнительная информация об этих файлах приведена в разделе Using .bbappend Files in Your Layer [4].

2.3. Изменение существующего задания

Во многих случаях можно изменить имеющееся задание linux-yocto в соответствии с вашими требованиями. Каждый выпуск YP включает новые задания для ядра Linux, из которых можно выбирать. Эти задания размещаются в ветви meta/recipes-kernel/linux каталога исходных кодов.

Изменение имеющегося задания может включать:

- создание файла добавления;
- применение исправлений;
- изменение конфигурации.

Перед внесением изменений в имеющееся задание создайте простой уровень, из которого вы сможете работать (см. 2.2. Создание и подготовка уровня).

2.3.1. Создание файла добавления

Этот файл вы создаёте на своём уровне и называете его на основе используемого задания linux-yocto. Например, при изменении задания meta/recipes-kernel/linux/linux-yocto_4.12.bb файл добавления обычно будет размещаться на вашем уровне с именем

```
your-layer/recipes-kernel/linux/linux-yocto_4.12.bbappend
```

Файлу добавления сначала следует расширить путь поиска FILESPATH путём добавления в переменную FILESEXTRAPATHS каталогов, где размещаются ваши файлы, как показано ниже

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

В примере путь ${THISDIR}/${PN} преобразуется в каталог linux-yocto в текущем каталоге. При добавлении новых файлов, которые меняют задания для ядра, и наличии преобразованного, как указано выше, пути, эти файлы нужно размещать в каталоге

```
your-layer/recipes-kernel/linux/linux-yocto/
```

Для работы с BSP для новой системы следует ознакомиться с документом [5].

В качестве примера рассмотрим файл, используемый BSP в составе уровня meta-yocto-bsp meta-yocto-bsp/recipes-kernel/linux/linux-yocto_4.12.bbappend. Содержимое этого файла приведено ниже. Отметим, что реальные идентификаторы представления могут быть иными в вашем уровне meta-yocto-bsp.

```
KBRANCH_genericx86  = "standard/base"
KBRANCH_genericx86-64  = "standard/base"

KMACHINE_genericx86 ?= "common-pc"
```

```
KMACHINE_genericx86-64 ?= "common-pc-64"
KBRANCH_edgerouter = "standard/edgerouter"
KBRANCH_beaglebone = "standard/beaglebone"
KBRANCH_mpc8315e-rdb = "standard/fsl-mpc8315e-rdb"

SRCREV_machine_genericx86    ?= "d09f2ce584d60ecb7890550c22a80c48b83c2e19"
SRCREV_machine_genericx86-64 ?= "d09f2ce584d60ecb7890550c22a80c48b83c2e19"
SRCREV_machine_edgerouter ?= "b5c8cfda2dfe296410d51e131289fb09c69e1e7d"
SRCREV_machine_beaglebone ?= "b5c8cfda2dfe296410d51e131289fb09c69e1e7d"
SRCREV_machine_mpc8315e-rdb ?= "2d1d010240846d7bff15d1fcc0cb6eb8a22fc78a"

COMPATIBLE_MACHINE_genericx86 = "genericx86"
COMPATIBLE_MACHINE_genericx86-64 = "genericx86-64"
COMPATIBLE_MACHINE_edgerouter = "edgerouter"
COMPATIBLE_MACHINE_beaglebone = "beaglebone"
COMPATIBLE_MACHINE_mpc8315e-rdb = "mpc8315e-rdb"

LINUX_VERSION_genericx86 = "4.12.7"
LINUX_VERSION_genericx86-64 = "4.12.7"
LINUX_VERSION_edgerouter = "4.12.10"
LINUX_VERSION_beaglebone = "4.12.10"
LINUX_VERSION_mpc8315e-rdb = "4.12.10"
```

Этот файл добавления служит для поддержки нескольких BSP из комплекта YP. Файл указывает машины в переменных COMPATIBLE_MACHINE, а для отображения имени машины на имя, используемое системой OE при сборке ядра Linux Yocto, указывает переменная переменная KMACHINE. Необязательные переменные KBRANCH задают используемую при сборке ветвь ядра.

В приведённом примере переменная KERNEL_FEATURES не используется, но она позволяет включить функции для конкретного ядра. Файл добавления указывает конкретные фиксации (commit) в репозитории Source Directory Git и ветвях репозитория метаданных для точного указания ядра, нужного для сборки BSP.

В рассматриваемом BSP отсутствует один элемент, который обычно требуется при разработке BSP, - файл конфигурации ядра (.config) для BSP. При создании BSP у вас скорей всего будет один или несколько конфигурационных файлов, определяющих параметры ядра для BSP. Можно задать конфигурацию поместив конфигурационные файлы в каталог, расположенный на том же уровне, где находится файл добавления для ядра, и имеющий такое же имя, которое задано для основного файла задания ядра. При выполнении этих условий достаточно просто указать файлы конфигурации оператором SRC_URI в файле добавления.

Предположим, например, что имеется набор опций конфигурации в файле network_configs.cfg. Можно поместить этот файл в каталог linux-yocto, а затем добавить оператор SRC_URI в файл добавления. Когда система OE будет собирать ядро, она найдёт и применит конфигурационные опции.

```
SRC_URI += "file://network_configs.cfg"
```

Для группировки связанных параметров в отдельные файлы применяется аналогичный подход. Ниже приведен пример группы раздельных конфигураций для Ethernet и графики в отдельных файлах с добавлением конфигурации с помощью операторов SRC_URI.

```
SRC_URI += "file://myconfig.cfg \
            file://eth.cfg \
            file://gfx.cfg"
```

Другой переменной, которая может применяться в файле добавления для задания ядра, является FILESEXTRAPATHS. Оператор позволяет указать каталоги, в которых система сборки OE будет искать файлы и исправления в процессе обработки задания.

Существуют и другие методы группировки и задания параметров конфигурации. Например, при работе с локальной копией репозитория ядра можно выбрать метаветвь ядра, внести изменения и передать их в локальный пустой клон ядра. В результате это будет напрямую добавлять конфигурационные опции в метаветвь для вашего BSP. Параметры конфигурации скорей всего окажутся в этом месте в любом случае при добавлении BSP в YP.

Однако в общем случае служба поддержки YP старается переносить параметры конфигурации, заданные SRC_URI, в метаветвь ядра. Это не только позволяет разработчикам BSP не беспокоится о размещении этих конфигураций в ветвях, но и даёт службе поддержки возможность применять «глобальные» знания о параметрах конфигурации, используемых множеством BSP в дереве. Это позволяет переносить базовые конфигурации в общие свойства.

2.3.2. Применение исправлений

Если у вас имеется одно исправление или небольшой их набор, которые вы хотите применить к исходным кодам ядра Linux, их можно применить как и для других заданий. Сначала patch-файлы копируются в каталог, добавленный оператором FILESEXTRAPATHS в ваш файл .bbappend, как описано в предыдущем параграфе, затем они указываются операторами SRC_URI.

Например, можно применить три набора исправления, добавив приведённые ниже строки в файл linux-yocto .bbappend на вашем уровне.

```
SRC_URI += "file://0001-first-change.patch"
SRC_URI += "file://0002-second-change.patch"
SRC_URI += "file://0003-third-change.patch"
```

При следующем запуске BitBake для сборки ядра Linux изменение задания будет обнаружено и все указанные исправления будут применены к ядру перед его сборкой.

Более подробная информация о применении исправлений к ядру приведена в параграфах 2.4. Использование devtool для применения изменений к ядру и 2.5. Традиционные методы внесения изменений в ядро.

2.3.3. Изменение конфигурации ядра

Можно вносить изменения в конфигурационный файл ядра .config, применяемый при сборке проекта, путём включения файла defconfig и указания фрагментов конфигурации в SRC_URI [1] для применения к этому файлу.

Если имеется работоспособный файл конфигурации ядра Linux .config, который вы хотите использовать, достаточно скопировать этот файл в нужный каталог ${PN} на уровне recipes-kernel/linux и переименовать его в defconfig. Затем добавьте в файл linux-yocto .bbappend на этом уровне строки

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://defconfig"
```

Значение SRC_URI говорит системе сборки, как найти файл, а FILESEXTRAPATHS расширяет переменную FILESPATH (каталоги поиска), включая в неё каталог ${PN} для сохранения изменений в конфигурации.

Система сборки применяет конфигурацию из файла defconfig до применения последующих фрагментов конфигурации. Окончательная конфигурация ядра будет определяться комбинацией параметров в файле defconfig и предоставленных фрагментах конфигурации. Следует помнить, что фрагменты конфигурации система сборки применяет после defconfig и они могут менять заданную этим файлом конфигурацию.

В общем случае предпочтительным вариантом изменения конфигурации является создание соответствующего фрагмента. Например, если нужно добавить базовую поддержку последовательной консоли, можно создать в каталоге ${PN} файл 8250.cfg с приведёнными ниже строками:

```
CONFIG_SERIAL_8250=y
CONFIG_SERIAL_8250_CONSOLE=y
CONFIG_SERIAL_8250_PCI=y
CONFIG_SERIAL_8250_NR_UARTS=4
CONFIG_SERIAL_8250_RUNTIME_UARTS=4
CONFIG_SERIAL_CORE=y
CONFIG_SERIAL_CORE_CONSOLE=y
```

Далее включается этот фрагмент конфигурации и расширяется переменная FILESPATH в файле .bbappend, как показано ниже.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://8250.cfg"
```

При следующем запуске BitBake для сборки ядра Linux изменения в задании будут обнаружены, извлечены и применены к новой конфигурации ядра для сборки.

Более подробная информация приведена в разделе 2.6. Настройка конфигурации ядра.

2.3.4. Файл defconfig в дереве кода

Может оказаться желательной настройка конфигурации ядра из файла defconfig, размещённого в дереве исходных кодов ядра для целевой системы. По умолчанию система сборки OE ищет файлы defconfig на уровне, используемом для Metadata, который находится «вне дерева» и указывает их с помощью строк

```
SRC_URI += "file://defconfig"
```

Если вы не хотите держать копии файлов defconfig на своём уровне, а разрешаете пользователям применять заданную по умолчанию конфигурацию из дерева ядра, сохраняя возможность добавлять изменения с помощью SRC_URI (например, путём добавления фрагментов), можно указать системе сборки OE использование файла defconfig, размещённого «в дереве ядра».

```
KBUILD_DEFCONFIG_KMACHINE ?= defconfig_file
```

Ниже приведен пример добавления переменной KBUILD_DEFCONFIG с common-pc и указания пути к файлу defconfig в дереве кода

```
KBUILD_DEFCONFIG_common-pc ?= "/home/scottrif/configfiles/my_defconfig_file"
```

Помимо изменения вашего задания для ядра и предоставления файла defconfig нужно убедиться, что никакие файлы или операторы SRC_URI не используют других файлов defconfig (например, linux-machine.inc). Иными словами, при обнаружении машиной сборки операторов, указывающих файл defconfig вне дерева кода, эти операторы будут переопределять переменную KBUILD_DEFCONFIG. Описание переменной KBUILD_DEFCONFIG приведено в [1].

2.4. Использование devtool для применения изменений к ядру

Этапы этой процедуры показывают применение файлов исправлений (patch) к ядру с помощью SDK и devtool.

Перед выполнением процедуры убедитесь в том, что выполнены все этапы, указанные в параграфе 2.1.1. Подготовка к работе с использованием devtool.

Применение исправлений к ядру включает изменение или добавление параметров конфигурации имеющегося ядра, изменение или добавление заданий, нужных для поддержки конкретных аппаратных функций и даже изменение кода самого ядра.

Приведённый ниже пример включает простое изменение, заключающееся в добавлении вывода на консоль эмулятора QEMU в процессе загрузки с помощью функции printk в файле calibrate.c исходного кода ядра. Внесение изменений и загрузка с новым образом обеспечивают вывод на консоль эмулятора дополнительной информации. Пример является продолжением процедуры установки, описанной в параграфе 2.1.1. Подготовка к работе с использованием devtool.

1. *Проверка файлов с исходным кодом ядра.* Сначала нужно использовать devtool для получения исходных кодов ядра. Убедитесь, что ваш терминал настроен на работу с SDK (см. 2.1.1. Подготовка к работе с использованием devtool).

   Для проверки кода используйте команду devtool, как показано ниже

   ```
   $ devtool modify linux-yocto
   ```

   Операция проверки имеет дефект, который может вызывать ошибку и показанное ниже сообщение

```
ERROR: Taskhash mismatch 2c793438c2d9f8c3681fd5f7bc819efa versus
       be3a89ce7c47178880ba7bf6293d7404 for
       /path/to/esdk/layers/poky/meta/recipes-kernel/linux/linux-yocto_4.10.bb.do_unpack
```

Это сообщение можно игнорировать.

2. *Редактирование файлов с исходным кодом.* При внесении простых изменений в файлы следуйте описанной ниже процедуре.

   a. *Смена рабочего каталога.* Вывод предыдущего этапа указывает, где размещены файлы исходного кода (например, ~/poky_sdk/workspace/sources/linux-yocto). Перейдите в нужный каталог, прежде чем вносить изменения в файл calibrate.c.

   ```
   $ cd ~/poky_sdk/workspace/sources/linux-yocto
   ```

   b. *Редактирование файла* init/calibrate.c с включением представленного ниже текста.

   ```
   void calibrate_delay(void)
   {
       unsigned long lpj;
       static bool printed;
       int this_cpu = smp_processor_id();

       printk("*************************************\n");
       printk("*                                   *\n");
       printk("*        HELLO YOCTO KERNEL         *\n");
       printk("*                                   *\n");
       printk("*************************************\n");

      if (per_cpu(cpu_loops_per_jiffy, this_cpu)) {
            .
            .
            .
   ```

3. *Сборка обновлённого ядра.* Для сборки ядра используется команда devtool

   ```
   $ devtool build linux-yocto
   ```

4. *Создание образа с новым ядром.* Новый образ создаётся с помощью команды devtool build-image.

   ```
   $ cd ~
   $ devtool build-image core-image-minimal
   ```

   Если исходный образ был представлен файлом Wic, можно воспользоваться другим методом создания обновлённого образа. Пример использования этого метода приведен на странице TipsAndTricks/KernelDevelopmentWithEsdk.

5. *Тестирование нового образа.* В нашем примере можно запустить новый образ с QEMU для проверки изменений.

   a. *Загрузка нового образа* в эмуляторе QEMU

   ```
   $ runqemu qemux86
   ```

   b. *Проверка изменений.* Войдите в систему с именем root без пароля и введите приведённую ниже команду для просмотра вывода при загрузке.

   ```
   # dmesg | less
   ```

   Вывод должен содержать сообщения printk.

6. *Фиксация изменений.* На консоли eSDK перейдите в каталог, где расположен изменённый файл calibrate.c и используйте приведённые ниже команды Git для фиксации внесённых изменений.

   ```
   $ cd ~/poky_sdk/workspace/sources/linux-yocto
   $ git status
   $ git add init/calibrate.c
   $ git commit -m "calibrate: Add printk example"
   ```

7. *Экспорт исправлений и создание файла добавления.* Для экспорта ваших изменений в качестве исправления (patche) и создания файла .bbappend file используйте приведённую ниже команду в консоли eSDK. В примере используется созданный ранее уровень meta-mylayer (см. п. 3 в параграфе 2.1.1. Подготовка к работе с использованием devtool).

   ```
   $ devtool finish linux-yocto ~/meta-mylayer
   ```

   После выполнения этой команды исправления и файл .bbappend будут находиться в каталоге ~/meta-mylayer/recipes-kernel/linux.

8. *Сборка образа с обновлённым ядром.* Сейчас можно собрать новый образ с учётом внесённых изменений. Для этого нужно выполнить приведённые ниже команды из каталога сборки в консоли BitBake.

   ```
   $ cd ~/poky/build
   $ bitbake core-image-minimal
   ```

2.5. Традиционные методы внесения изменений в ядро

Описанная здесь процедура показывает применение изменений (patch) к файлам ядра с использованием традиционных методов (т. е. без devtool и eSDK, описанных выше). Перед началом работы убедитесь в готовности системы к обновлению ядра (см. параграф 2.1.2. Подготовка к традиционной работе с ядром).

Внесение изменений в ядро включает изменение или добавление конфигурации имеющегося ядра, изменение или добавления заданий, требуемых для поддержки определённого оборудования или функций, а также изменение файлов исходного кода.

Пример в этом параграфе включает простое добавление вывода в консоль эмулятора QEMU при загрузке с помощью функции printk в файле ядра calibrate.c. Внесение этих изменений и загрузка нового образа обеспечат вывод на

консоль дополнительной информации. Пример является продолжением процедуры настройки, описанной в параграфе 2.1.2. Подготовка к традиционной работе с ядром.

1. *Редактирование файлов с исходным кодом.* Перед выполнением этого этапа убедитесь в наличии локальной копии Git-репозитория исходных кодов ядра (предполагается установка в соответствии с параграфом 2.1.2. Подготовка к традиционной работе с ядром).

   a. *Смена рабочего каталога.* Перейдите в каталог локальной копии репозитория Git с файлами ядра. В нашем примере для этого служит команда

   ```
   $ cd ~/linux-yocto-4.12/init
   ```

   b. *Редактирование файла* calibrate.c, в соответствии с приведённым ниже текстом.

   ```
   void calibrate_delay(void)
   {
       unsigned long lpj;
       static bool printed;
       int this_cpu = smp_processor_id();

       printk("*************************************\n");
       printk("*                                   *\n");
       printk("*        HELLO YOCTO KERNEL         *\n");
       printk("*                                   *\n");
       printk("*************************************\n");

      if (per_cpu(cpu_loops_per_jiffy, this_cpu)) {
            ...
   ```

2. *Фиксация и представление изменений.* С помощью стандартных команд Git представьте внесённые правки.

   ```
   $ git add calibrate.c
   $ git commit -m "calibrate.c - Added some printk statements"
   ```

   Если этого не сделать, система сборки OE не учтёт этих изменений.

3. *Обновление файла local.conf для указания файлов исходного кода.* В дополнение к указанию в файле local.conf использовать kernel-modules и машину qemux86 можно указать обновлённые файлы исходного кода ядра. Это делается с помощью операторов SRC_URI и SRCREV, как показано ниже.

   ```
   $ cd ~/poky/build/conf
   ```

   Добавляем приведённый ниже текст в файл local.conf.

```
SRC_URI_pn-linux-yocto = "git:///path-to/linux-yocto-4.12;protocol=file;name=machine;branch=standard/base; \
git:///path-to/yocto-kernel-cache;protocol=file;type=kmeta;name=meta;branch=yocto-4.12;destsuffix=${KMETA}"
SRCREV_meta_qemux86 = "${AUTOREV}"
SRCREV_machine_qemux86 = "${AUTOREV}"
```

   Не забудьте заменить *path-to* реальным именем каталога с локальным репозиторием Git. Нужно также корректно указать ветви и типы машин. В примере указана ветвь standard/base и машина qemux86.

4. *Сборка образа.* После правки исходного кода, фиксации и представления изменений и указания в файле local.conf файлов ядра можно использовать BitBake для сборки образа.

   ```
   $ cd ~/poky/build
   $ bitbake core-image-minimal
   ```

5. *Загрузка образа.* Загрузите обновленный образ в эмулятор QEMU с помощью приведённых ниже команд (для входа в систему используется имя root без пароля):

   ```
   $ cd ~/poky/build
   $ runqemu qemux86
   ```

6. *Просмотр изменений.* При загрузке QEMU можно увидеть сообщения, прокручивая экран вверх. Если прокрутка не поддерживается, воспользуйтесь командой

   ```
   # dmesg | less
   ```

   Вы увидите сообщения printk, выводимые в процессе загрузки.

7. *Генерация файла изменений.* После проверки корректности работы можно создать файл *.patch в каталоге с исходным кодом ядра.

   ```
   $ cd ~/linux-yocto-4.12/init
   $ git format-patch -1
   0001-calibrate.c-Added-some-printk-statements.patch
   ```

8. *Перенос файла изменений на ваш уровень.* Чтобы при последующих сборках учитывались внесённые изменений, нужно перенести созданный patch-файл на ваш уровень meta-mylayer. В нашем примере созданный ранее уровень размещается в домашнем каталоге и называется meta-mylayer. При создании уровня с помощью сценария yocto-create иерархия для поддержки исправлений не создаётся, поэтому перед переносом patch-файла нужно добавить соответствующие каталоги на вашем уровне, как показано ниже.

   ```
   $ cd ~/meta-mylayer
   $ mkdir recipes-kernel
   $ mkdir recipes-kernel/linux
   $ mkdir recipes-kernel/linux/linux-yocto
   ```

   После создания нужной иерархии можно переместить patch-файл с помощью команды

   ```
   $ mv ~/linux-yocto-4.12/init/0001-calibrate.c-Added-some-printk-statements.patch ~/meta-
   mylayer/recipes-kernel/linux/linux-yocto
   ```

9. *Создание файла добавления.* В заключение нужно создать файл linux-yocto_4.12.bbappend и добавить в него операторы, которые позволят системе сборки OE найти исправления. Этот файл должен размещаться в каталоге recipes-kernel/linux и называться linux-yocto_4.12.bbappend. Содержимое файла показано ниже.

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

```
SRC_URI_append = " file://0001-calibrate.c-Added-some-printk-statements.patch"
```

Операторы FILESEXTRAPATHS и SRC_URI позволяют системе сборки OE найти patch-файл.

Дополнительная информация о файлах дополнения и «заплатках» представлена в параграфах 2.3.1. Создание файла добавления и 2.3.2. Применение исправлений. Можно также обратиться к разделу Using .bbappend Files in Your Layer [4].

Для последующей сборки core-image-minimal и просмотра влияния изменений можно устранить временные файлы исходного кода в poky/build/tmp/work/... и побочные эффекты прежней сборки с помощью команд

```
$ cd ~/poky/build
$ bitbake -c cleanall yocto-linux
$ bitbake core-image-minimal -c cleanall
$ bitbake core-image-minimal
$ runqemu qemux86
```

2.6. Настройка конфигурации ядра

Настройка ядра в YP заключается в том, чтобы обеспечить наличие в файле .config всей нужной информации для создаваемого образа. Можно использовать инструмент menuconfig и фрагменты конфигурации, чтобы создать нужный файл .config или сохранить проверенную конфигурацию в файле defconfig, который система сборки будет использовать для настройки ядра.

В этом разделе описано использование menuconfig, создание и применение фрагментов конфигурации, а также изменение в интерактивном режиме файла .config с нужной конфигурацией. Конфигурация ядра описана также в разделе 2.3.3. Изменение конфигурации ядра.

2.6.1. Использование menuconfig

Простейшим способом задать конфигурацию ядра является инструмент menuconfig [2]. Это позволяет настраивать конфигурацию в интерактивном режиме с доступом к справке по конфигурационным параметрам. Для работы с menuconfig в среде YP требуется выполнить ряд условий, приведённых ниже

- Поскольку menuconfig запускается с использованием BitBake, нужно настроить рабочее окружение с помощью сценария oe-init-build-env в вашем каталоге сборки [2].
- Нужно проверить наличие конфигурации вашей сборки в каталоге исходных кодов [2].
- На сборочном хосте должны быть установлены пакеты[1]:

  libncurses5-dev

  libtinfo-dev

Приведённые ниже команды инициализируют окружение BitBake, запускают задачу do_kernel_configme [2] и инструмент menuconfig. Предполагается что каталог верхнего уровня в Source Directory имеет имя ~/poky.

```
$ cd poky
$ source oe-init-build-env
$
$ bitbake linux-yocto -c menuconfig
```

После запуска menuconfig на экране появится стандартный интерфейс настройки конфигурационных параметров ядра. Внесите нужные изменения и завершите работу с сохранением результатов в конфигурационном файле .config.

Файл .config можно использовать в качестве defconfig (см. параграфы 2.3.3. Изменение конфигурации ядра, 2.3.4. Файл defconfig в дереве кода и 2.6.2. Создание файла defconfig).

Рассмотрим пример настройки параметров CONFIG_SMP для ядра проекта linux-yocto-4.12. Система сборки OE распознает это ядро как linux-yocto по метаданным (например, PREFERRED_VERSION_linux-yocto ?= "12.4%").

После запуска menuconfig установите нужные параметры конфигурации ядра и сохраните файл. В нашем примере отключается опция CONFIG_SMP путём снятия отметки Symmetric Multi-Processing Support (клавиша N). После обновления файла .config система сборки будет использовать его для настройки ядра в процессе сборки. Файл можно посмотреть в каталоге tmp/work/. Реальный файл .config для использования при сборке будет размещаться в дереве исходных кодов ядра. Например, при сборке ядра Linux Yocto на основе linux-yocto-4.12 для создания образа QEMU с архитектурой x86 файл .config будет находиться в каталоге poky/build/tmp/work/qemux86-poky-linux/linux-yocto/4.12.12+gitAUTOINC+eda4d18...967-r0/linux-qemux86-standard-build/.config. Имя каталога указано не полностью с целью сокращения. Кроме того, часть пути может меняться в зависимости от собираемого ядра.

В файле .config указаны параметры конфигурации ядра. В нашем примере будет строка, отключающая поддержку симметричной многопроцессорной обработки

```
# CONFIG_SMP is not set
```

Хорошим методом изоляции изменений конфигурации является использование комбинации menuconfig и простых команд оболочки. Перед изменением конфигурации с помощью menuconfig скопируйте файл .config и переименуйте копию (чтобы понять, какая конфигурация находится в файле), а затем используйте menuconfig для внесения требуемых изменений. После сохранения можно сравнить новый файл с прежним. Различия между файлами можно использовать для создания фрагментов конфигурации на вашем уровне ядра.

[1] В некоторых дистрибутивах Linux имена пакетов могут отличаться. Например, в Mageia первый называется libncurses-devel (lib64ncurses-devel) и включает в себя второй.

2.6.2. Создание файла defconfig

Файл defconfig в контексте YP зачастую является копией файла .config из каталога сборки или дерева исходных кодов ядра, перенесённой в пространство задания. Файл defconfig можно использовать для сохранения проверенной конфигурации ядра и система сборки OE будет создавать из него финальный файл .config.

YP без готового проекта не включает файлов defconfig и .config. Финальный файл .config, используемый для настройки ядра, создаёт система сборки OE.

Для создания файла defconfig следует взять полный файл .config рабочей конфигурации ядра Linux и скопировать его в подходящий каталог ${PN} в дереве recipes-kernel/linux вашего уровня (например, ~/meta-mylayer/recipes-kernel/linux/linux-yocto/defconfig), а затем переименовать в defconfig. После этого в файл linux-yocto .bbappend на вашем уровне нужно добавить строки

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://defconfig"
```

Значение SRC_URI говорит системе сборки, как найти файл, а FILESEXTRAPATHS расширяет переменную FILESPATH (каталоги поиска), включая в неё каталог ${PN} для сохранения изменений в конфигурации.

Система сборки применяет конфигурацию из файла defconfig до применения последующих фрагментов конфигурации. Окончательная конфигурация ядра будет определяться комбинацией параметров в файле defconfig и предоставленных фрагментах конфигурации. Следует помнить, что фрагменты конфигурации система сборки применяет после defconfig и они могут менять заданную этим файлом конфигурацию (см. 2.3.3. Изменение конфигурации ядра).

2.6.3. Создание фрагментов конфигурации

Фрагменты конфигурации являются просто опциями ядра, указанными в файлах, размещённых там, где система сборки OE может найти и применить их. Эти фрагменты применяются после рассмотрения опций, заданных в файле defconfig. Таким образом, конфигурация ядра определяется набором опций в файле defconfig с учётом дополнений и изменений, вносимых фрагментами конфигурации. Синтаксически фрагменты конфигурации идентичны содержимому файла .config в каталоге сборки (см. параграф 2.6.1. Использование menuconfig).

Создание фрагментов конфигурации достаточно просто. Одним из вариантов является использование команд системной оболочки. Например, можно добавить опцию в имеющийся фрагмент my_smp.cfg с помощью команды

```
$ echo "CONFIG_SMP=y" >> my_smp.cfg
```

Все файлы с фрагментами конфигурации должны иметь расширение .cfg, чтобы система сборки OE могла распознавать их.

Другим методом создания фрагментов конфигурации является использование различий между двумя конфигурационными файлами, один из которых создан ранее и сохранен, а другой недавно создан с помощью menuconfig.

1. *Начальная настройка конфигурации ядра* с помощью команды

   ```
   $ bitbake linux-yocto -c kernel_configme -f
   ```

   Это обеспечит создание файла .config по известным источникам. В некоторых ситуациях состояние сборки может быть не известно, поэтому лучше выполнять эту команду до запуска menuconfig.

2. *Настройка конфигурации с помощью menuconfig*

   ```
   $ bitbake linux-yocto -c menuconfig
   ```

3. *Создание фрагмента конфигурации.* Введите команду diffconfig для создания фрагмента конфигурации fragment.cfg, который будет помещён в каталог ${WORKDIR}

   ```
   $ bitbake linux-yocto -c diffconfig
   ```

Команда diffconfig создаёт файл со списком назначений переменных конфигурации ядра CONFIG_. Использование вывода программы для создания фрагмента конфигурации описано в параграфе 2.3.3. Изменение конфигурации ядра. Этот метод можно использовать и при создании фрагментов конфигурации для BSP (см. 3.3.5. Описания BSP).

Файлы с фрагментами конфигурации можно разместить в области, указанной переменной SRC_URI, как это задано в файле bblayers.conf, размещённом в каталоге вашего уровня. Система сборки OE заберёт эти фрагменты и добавит в конфигурацию ядра. Предположим, например, размещение некоторых опций конфигурации в файле myconfig.cfg. Если файл размещён в каталоге linux-yocto, который находится в одном каталоге с файлом добавления для ядра и в этот файл включены приведённые ниже операторы, конфигурационные опции будут найдены и применены при сборке ядра.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://myconfig.cfg"
```

Как отмечено выше, можно сгруппировать связанные конфигурации в несколько файлов и указать их в операторе SRC_URI. Например, можно сделать разные фрагменты конфигурации для Ethernet и графики, указав их в операторе SRC_URI внутри файла дополнения.

```
SRC_URI += "file://myconfig.cfg \
       file://eth.cfg \
       file://gfx.cfg"
```

2.6.4. Проверка корректности конфигурации

Для проверки конфигурации можно использовать задачу do_kernel_configcheck с помощью команды

```
$ bitbake linux-yocto -c kernel_configcheck -f
```

При работе этой задачи выдаются предупреждения об отсутствии запрошенной конфигурации в финальном файле .config или переопределении правил конфигурации во фрагменте аппаратной конфигурации.

Для запуска этой задачи в системе сборки должен быть подготовленный файл .config. Создание конфигурационного файла описано в параграфе 2.6.1. Использование menuconfig.

Ниже приведен пример вывода задачи do_kernel_configcheck.

```
     Loading cache: 100% |#########################################################| Time: 0:00:00
     Loaded 1275 entries from dependency cache.
     NOTE: Resolving any missing task queue dependencies

     Build Configuration:
         .
         .
         .

     NOTE: Executing SetScene Tasks
     NOTE: Executing RunQueue Tasks
     WARNING: linux-yocto-4.12.12+gitAUTOINC+eda4d18ce4_16de014967-r0 do_kernel_configcheck:
         [kernel config]: specified values did not make it into the kernel's final configuration:

     ---------- CONFIG_X86_TSC -----------------
     Config: CONFIG_X86_TSC
     From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
bsp/common-pc/common-pc-cpu.cfg
     Requested value:  CONFIG_X86_TSC=y
     Actual value:


     ---------- CONFIG_X86_BIGSMP -----------------
     Config: CONFIG_X86_BIGSMP
     From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
cfg/smp.cfg
           /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
defconfig
     Requested value:  # CONFIG_X86_BIGSMP is not set
     Actual value:


     ---------- CONFIG_NR_CPUS -----------------
     Config: CONFIG_NR_CPUS
     From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
cfg/smp.cfg
           /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
bsp/common-pc/common-pc.cfg
           /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
defconfig
     Requested value:  CONFIG_NR_CPUS=8
     Actual value:     CONFIG_NR_CPUS=1


     ---------- CONFIG_SCHED_SMT -----------------
     Config: CONFIG_SCHED_SMT
     From: /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
cfg/smp.cfg
           /home/scottrif/poky/build/tmp/work-shared/qemux86/kernel-source/.kernel-meta/configs/standard/
defconfig
     Requested value:  CONFIG_SCHED_SMT=y
     Actual value:



     NOTE: Tasks Summary: Attempted 288 tasks of which 285 didn't need to be rerun and all succeeded.

     Summary: There were 3 WARNING messages shown.
```

Вывод описывает различные проблемы, которые могут возникать, а также указывает элементы с нарушениями конфигурации. Эту информацию можно использовать для корректировки конфигурационных файлов и повтора задач do_kernel_configme и do_kernel_configcheck, пока проблемы не будут устранены.

Дополнительная информация о настройке конфигурации приведена в параграфе 2.6.1. Использование menuconfig.

2.6.5. Тонкая настройка конфигурации ядра

Вы можете проверить корректность и эффективность файла .config, просматривая вывод аудита фрагментов конфигурации ядра, отмечая проблемы и внося изменения, а затем повторяя процедуру проверки.

В процессе сборки ядра выполняется задача do_kernel_configcheck, проверяющая конфигурацию ядра путём сравнения финального файла .config со входными файлами. В процессе проверки выдаются предупреждения в случае наличия перечисленных ниже проблем.

- Запрошенная опция не включена в финальный файл .config.
- Опция указана более одного раза в одном фрагменте конфигурации.
- Был переопределен элемент конфигурации, помеченный как обязательный (required).
- Для платы переопределена опция, не относящаяся к ней.
- Указанные опции не пригодны для собираемого ядра (недопустимо их присутствие где-либо).

Задача do_kernel_configcheck может также сообщать о переопределении опций в процессе обработки.

В каждом выводимом предупреждении указывается файл, содержащий список опций и указатель на фрагмент конфигурации, где он определён. Для упрощения настройки используйте перечисленные ниже операции.

1. *Используйте рабочую конфигурацию.* Начните с полной конфигурации, которая заведомо работает (проверена в плане сборки ядра и его загрузки).
2. *Запустите задачи настройки и проверки конфигурации.* Поочерёдно запустите задачи do_kernel_configme и do_kernel_configcheck, как показано ниже

```
$ bitbake linux-yocto -c kernel_configme -f
$ bitbake linux-yocto -c kernel_configcheck -f
```

3. *Обработайте результаты.* Возьмите список файлов из предупреждений задачи do_kernel_configcheck и выполните перечисленные ниже операции.
   - Удалите значения, которые переопределены во фрагменте, но не меняют финальный файл .config.
   - Проанализируйте и при необходимости удалите из файла .config опции, которые переопределяют требуемые настройки.
   - Проанализируйте и при необходимости удалите не относящиеся к вашей плате настройки.
   - Удалите повторяющиеся и недействительные опции.
4. *Повторите задачи настройки и проверки конфигурации.* После обработки результатов аудита конфигурации ядра запустите повторно задачи do_kernel_configme и do_kernel_configcheck для просмотра результатов своих правок. Если проблемы остались, вернитесь к п. 3.

Выполнение этой процедуры до устранения всех проблем позволит получить оптимальный файл конфигурации, после чего можно запустить процесс сборки ядра Linux.

2.7. Извлечение значений переменных

Иногда полезно бывает определить, во что преобразуется переменная в процессе сборки. Можно проверить значения переменных путём просмотра вывода команды bitbake -e. Команда выводит достаточно много информации, поэтому лучше направить её в текстовый файл, которым можно будет просмотреть.

```
$ bitbake -e virtual/kernel > some_text_file
```

Этот файл будет содержать значения каждой переменной, преобразуемой и используемой системой сборки OE.

2.8. Работа с «грязной» строкой версии ядра

Если при сборке ядра строка версии имеет суффикс +dirty или -dirty, это говорит о незафиксированных изменениях в каталоге исходных кодов ядра. Для очистки строки версии используйте описанные ниже процедуры.

1. *Поиск не зафиксированных изменений.* Перейдите в каталог локального репозитория Git (каталог исходных кодов) и найдите изменённые, добавленные или удалённые файлы с помощью команды

```
$ git status
```

2. *Зафиксируйте изменения.* Найденные изменения следует зафиксировать (commit) в дереве исходных кодов ядра независимо от того, планируется ли сохранение, экспорт или использование внесённых изменений.

```
$ git add
$ git commit -s -a -m "getting rid of -dirty"
```

3. *Повторная сборка ядра.* После фиксации изменений следует заново собрать ядро.

   В зависимости от конкретного процесса разработки ядра команда повторной сборки может меняться. Информация о сборке ядра с помощью devtool представлена в параграфе 2.4. Использование devtool для применения изменений к ядру, а для использования традиционных методов - в параграфе 2.5. Традиционные методы внесения изменений в ядро.

2.9. Работа со своими исходными кодами

Если вы не можете работать с одной из версий ядра, поддерживаемых имеющимися заданиями linux-yocto, вы можете применять инструменты YP для работы со своими исходными кодами ядра. В таких случаях вы не сможете воспользоваться имеющимися метаданными ядра и результатами стабилизации исходных кодов linux-yocto. Однако это не помешает управлять своими метаданными в том же формате, который применяется в linux-yocto. Поддержка совместимых форматов облегчит слияние с linux-yocto в будущей версии ядра с взаимной поддержкой.

Чтобы помочь при работе со своими исходными кодами, YP предоставляет специальное задание linux-yocto (linux-yocto-custom.bb), использующее исходные коды с сайта kernel.org и инструменты для управления метаданными ядра. Это задание можно найти в репозитории poky Git репозитория исходных кодов YP

```
poky/meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb
```

Ниже описаны базовые этапы работы со своим деревом исходных кодов

1. *Создание копии задания для ядра.* Скопируйте задание linux-yocto-custom.bb на свой уровень и дайте файлу осмысленное имя. В имя следует включить используемую версию ядра Yocto Linux (например, linux-yocto-myproject_4.12.bb, где 4.12 указывает номер базовой версии ядра Linux, с которой вы будете работать).
2. *Создание каталога для файлов с исправлениями.* В том же каталоге внутри своего уровня создайте соответствующий каталог для хранения ваших файлов изменений (patch) и конфигурационных файлов (например, linux-yocto-myproject).
3. *Обеспечение наличия конфигурации.* Убедитесь в наличии файла defconfig или файлов с фрагментами конфигурации на своём уровне. При использовании задания linux-yocto-custom.bb нужно будет задать конфигурацию. Если файла defconfig нет, введите из каталога с исходными кодами ядра команду

```
$ make defconfig
```

   После завершения работы команды скопируйте полученный файл .config в каталог files на своём уровне, переименуйте его в defconfig и добавьте в переменную SRC_URI для задания.

Выполнение команды make defconfig создаёт конфигурацию с принятыми по умолчанию опциями для вашей архитектуры, определёнными вашим ядром. Однако пригодность конфигурации для вашего конкретного случая не гарантируется. Особенно велика вероятность возникновения проблем для архитектуры, отличающейся от x86. Для использования файлов в такой архитектуре нужно указать более конкретно вашу плату (например, для arm следует заглянуть в каталог arch/arm/configs и использовать наиболее подходящий вариант defconfig в качестве стартовой конфигурации).

4. *Редактирование задания.* Отредактируйте в своём задании перечисленные ниже переменные с учётом потребностей проекта.
   - SRC_URI. В этой переменной следует указать репозиторий Git, который использует один из поддерживаемых протоколов сборщика Git (file, git, http и т. п.). В SRC_URI следует также указать defconfig или файлы фрагментов конфигурации. Образец задания включает пример SRC_URI.
   - LINUX_VERSION - используемая версия ядра Linux (например, 4.12).
   - LINUX_VERSION_EXTENSION - значение опции CONFIG_LOCALVERSION, которое будет включено в собираемое ядро и доступно по команде uname.
   - SRCREV - идентификатор фиксации (commit ID), из которой будет выполняться сборка.
   - PR - номер сборки, увеличиваемый каждый раз для указания системе OE факта изменения задания.
   - PV - принятое по умолчанию значение PV обычно адекватно. Оно комбинирует LINUX_VERSION с выпуском SCM1 из переменной SRCPV и даёт строку вида

     ```
     3.19.11+git1+68a635bf8dfb64b02263c1ac80c948647cc76d5f_1+218bd8d2022b9852c60d32f0d770931e3cf343e2
     ```

     Большой размер строки PV позволяет гарантировать использование именно тех источников, из которых планируется собирать задание.
   - COMPATIBLE_MACHINE - список машин, поддерживаемых новым заданием. Для этой переменной в примере по умолчанию используется регулярное выражение, которому соответствует лишь пустая строка "(^$)". Это вызывает явный отказ при сборке, поэтому переменную нужно изменить в соответствии со списком машин, которые будет поддерживать новое задание. Например, поддержку машин qemux86 и qemux86-64 можно указать в форме

     ```
     COMPATIBLE_MACHINE = "qemux86|qemux86-64"
     ```
5. *Настройка вашего задания.* Выполните подобающую настройку своего задания, как описано для имеющегося задания linux-yocto (см. параграф 2.3. Изменение существующего задания).

2.10. Работа с модулями вне дерева исходных кодов

В этом разделе описаны этапы сборки внешних (out-of-tree) модулей на целевой платформе и встраивания таких модулей в сборку.

2.10.1. Сборка внешних модулей на целевой платформе

Хотя традиционная модель разработки YP будет включать модули ядра в обычный процесс сборки, может возникнуть потребность в сборке модуля на целевой платформе. Это возможно в тех случаях, когда платформа имеет производительность, достаточную для компиляции исходного кода. Однако перед выбором такого подхода следует оценить возможности использования кросс-компиляции модуля на сборочном хосте.

Если вы хотите собрать внешний модуль на целевой платформе, вам потребуется выполнить на ней несколько операций, описанных ниже. Пакет kernel-dev устанавливается по умолчанию во все образы *.sdk, а kernel-devsrc - на большинство таких образов. Однако потребуется создать некоторые сценарии до попытки собрать внешние модули на целевой платформе, использующей такой образ.

Перед попыткой собрать внешний модуль нужно войти в систему платформы как пользователь root и перейти в каталог /usr/src/kernel, а затем собрать сценарии.

```
# cd /usr/src/kernel
# make scripts
```

Поскольку все задания для образов SDK включают dev-pkgs, пакет kernel-dev будет установлен как часть образа SDK, а пакеты kernel-devsrc - как часть соответствующих образов SDK. В SDK применяются сценарии для сборки внешних модулей. После перехода в указанный каталог и создания сценариев вы сможете собирать внешние модули.

2.10.2. Встраивание внешних модулей

Хотя всегда предпочтительней работать с кодами, встроенными в дерево ядра Linux, при необходимости собрать внешний модуль ядра можно использовать задание hello-mod.bb в качестве образца для подготовки нужного задания по сборке внешнего модуля. Упомянутый шаблон находится в репозитории Git YP Source Repository

```
poky/meta-skeleton/recipes-kernel/hello-mod/hello-mod_0.1.bb
```

Для начала работы скопируйте этот файл задания на свой уровень и дайте ему осмысленное имя (например, mymodule_1.0.bb). В том же каталоге создайте подкаталог с именем files, где будут храниться файлы исходного кода, исправления и другие файлы, требуемые для сборки модуля, не включённого в дерево исходных кодов ядра. Затем отредактируйте задание в соответствии с потребностями модуля. Обычно требуется установить значения переменных:

- DESCRIPTION;
- LICENSE*;
- SRC_URI;
- PV.

В зависимости от системы сборки, используемой для модулей, могут потребоваться некоторые дополнительные настройки. Например, типичный файл Makefile для модуля имеет вид, похожий на файл из шаблона hello-mod.

```
obj-m := hello.o

SRC := $(shell pwd)

all:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC)

modules_install:
    $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install
...
```

Важно отметить здесь переменную KERNEL_SRC. Класс module устанавливает для неё и переменной KERNEL_PATH значение ${STAGING_KERNEL_DIR} с необходимой для сборки модуля информацией о ядре Linux. Если в Makefile модуля используется другая переменная, вы можете переопределить её на этапе do_compile или создать файл исправления для Makefile, позволяющий работать с переменными KERNEL_SRC и KERNEL_PATH.

После подготовки задания вы наверняка захотите включить модуль в свои образы. Для этого следует ознакомиться с описаниями перечисленных ниже переменных в [1] и установить для них значения, подходящие для целевой машины.

- MACHINE_ESSENTIAL_EXTRA_RDEPENDS;
- MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS;
- MACHINE_EXTRA_RDEPENDS;
- MACHINE_EXTRA_RRECOMMENDS.

Модули часто не требуются при загрузке и могут быть исключены из некоторых конфигураций сборки. Для повышения гибкости используйте переменную

```
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-mymodule"
```

Значение переменной выводится путём добавления имени файла модуля без расширения .ko.

Поскольку переменная является RRECOMMENDS, а не RDEPENDS, сборка не будет завершаться ошибкой в результате невозможности включения модуля в образ.

2.11. Проверка изменений и фиксации

При работе с ядром часто возникает вопрос о том, какие изменения были внесены в дерево исходных кодов. Можно воспользоваться утилитой grep в рекурсивном режиме, но Git лучше подходит для просмотра или поиска изменений в дереве кодов ядра.

2.11.1. Поиск изменений в ядре

Ниже приведено несколько примеров использования команд Git для поиска изменений в ядре. В примерах предполагается, что до фиксации диапазона внесённых изменений история kernel.org смешана с историей изменений ядра в YP. Можно сформировать диапазоны изменений путём использования имён ветвей из дерева ядра в качестве верхнего и нижнего маркеров фиксации (commit) с помощью команд Git. Имена ветвей можно получить с помощью web-интерфейса репозитория YP на странице http://git.yoctoproject.org.

Для просмотра всего диапазона изменений используйте команду git whatchanged с указанием диапазона фиксации для ветви (commit..commit).

Ниже приведен пример просмотра изменений в ветви emenlow ядра linux-yocto-3.19. Нижним маркером служит фиксация, связанная с ветвью standard/base, а верхний маркер связан с фиксацией ветви standard/emenlow.

```
$ git whatchanged origin/standard/base..origin/standard/emenlow
```

Для просмотра кратких (в одну строку) сводок изменений служит команда git log

```
$ git log --oneline origin/standard/base..origin/standard/emenlow
```

Для просмотра различий в коде в результате изменений служит команда

```
$ git diff origin/standard/base..origin/standard/emenlow
```

Для просмотра сообщений в журнале фиксации и текста различий служит команда

```
$ git show origin/standard/base..origin/standard/emenlow
```

Приведённая ниже команда создаёт отдельный patch-файл для каждого изменения, размещая файлы для каждой фиксации в каталоге Documents.

```
$ git format-patch -o $HOME/Documents origin/standard/base..origin/standard/emenlow
```

2.11.2. Просмотр изменений конкретной функции или ветви

Теги в дереве ядра YP делят изменения конфигурации по значимым функциям или ветвям. Команда git show <tag> позволяет видеть изменения по тегам. Например, приведённая ниже команда покажет изменения для systemtap.

```
$ git show systemtap
```

Можно использовать команду git branch --contains <tag> для просмотра ветвей, содержащих определённую возможность. Например, приведённая ниже команда покажет ветви с возможностью systemtap.

```
$ git branch --contains systemtap
```

2.12. Добавление функций ядра recipe-space

Можно добавить функции ядра в пространство заданий recipe-space с помощью переменной KERNEL_FEATURES и указания пути к файлу .scc в операторе SRC_URI. При добавлении функций таким способом система сборки OE проверяет наличие всех функций и при отсутствии той или иной останавливает сборку. Функции ядра обрабатываются

после настройки и внесения изменений (patch). Поэтому добавление возможностей таким путём гарантирует наличие и включение соответствующих функций без необходимости полного аудита дополнений SRC_URI в других уровнях.

Возможность ядра добавляется как часть переменной KERNEL_FEATURES с указанием пути к файлу .scc относительно корня метаданных ядра. Система сборки OE ищет все формы метаданных ядра по оператору SRC_URI независимо от их размещения в кэше (kernel-cache), метаданных ядра системы или recipe-space (часть задания для ядра). Дополнительная информация о метаданных приведена в параграфе 3.4. Размещение метаданных ядра.

При указании файла возможностей .scc в операторе SRC_URI система сборки OE добавляет каталог с файлом .scc и все его подкаталоги в путь поиска функций ядра. Благодаря поиску в подкаталогах, достаточно указать один файл .scc в операторе SRC_URI для множества возможностей ядра.

Рассмотрим пример с добавлением возможности test.scc при сборке ядра.

1. *Создание файла возможности.* Создаётся файл .scc и указывается в операторе SRC_URI как другие файлы исправлений, .cfg или элементов сборки.

   - Каталог с файлом .scc должен быть включён в путь поиска сборщика, как это делается для файлов .patch.
   - Можно создать дополнительные файлы .scc в том же каталоге или его подкаталогах.

   Предположим, что функция test.scc добавлена в файл test.scc, как показано ниже.

   ```
   my_recipe
      |
      +-linux-yocto
         |
         +-test.cfg
         +-test.scc
   ```

   В этом примере каталог linux-yocto включает файл возможности test.scc и фрагмент конфигурации test.cfg.

2. *Добавление файла возможности в SRC_URI.* Включите файл .scc в оператор SRC_URI вашего задания

   ```
   SRC_URI_append = " file://test.scc"
   ```

   Пробел в начале значения важен, поскольку путь добавляется в конец имеющегося пути.

3. *Указание возможности как функции ядра.* Добавьте возможность в оператор KERNEL_FEATURES.

   ```
   KERNEL_FEATURES_append = " test.scc"
   ```

   Система сборки OE обработает эту возможность при сборке ядра.

   При наличии других возможностей «ниже» test.scc они размещаются в дереве каталога с файлом test.scc.

Глава 3. Работа с расширенными метаданными (yocto-kernel-cache)

3.1. Обзор

В дополнение к поддержке фрагментов конфигурации и patch-файлов инструменты YP также поддерживают обширный набор метаданных, которые можно применять для задания комплексных правил и поддержки BSP. Назначение Metadata и инструментов для поддержки заключается в помощи при управлении сложными конфигурациями и исходными кодами для поддержки множества BSP и типов ядер Linux.

Метаданные ядра (Kernel Metadata) размещаются во многих местах, одним из которых является репозиторий Git yocto-kernel-cache, указанный под заголовком Yocto Linux Kernel на странице Yocto Project Source Repositories.

Инструменты для работы с ядром (kern-tools) также размещены в этом репозитории под заголовком Yocto Linux Kernel. Задание для сборки этих инструментов размещено в файле meta/recipes-kernel/kern-tools/kern-tools-native_git.bb каталога исходных кодов (например, poky).

3.2. Использование метаданных ядра в задании

Как отмечено выше, YP включает метаданные ядра, размещённые в репозитории Git yocto-kernel-cache. Эти метаданные определяют пакеты BSP, которые соответствуют определениям в заданиях linux-yocto для BSP. Пакет BSP представляет собой совокупность правил для ядра и разрешённых (включённых) функций оборудования. На BSP можно влиять из задания linux-yocto.

Задание для ядра Linux с метаданными ядра (например, унаследованными из файла linux-yocto.inc) называют заданием в стиле linux-yocto. Каждое такое задание определяет переменную KMACHINE, значение которой обычно совпадает со значением переменной MACHINE, используемой BitBake. Однако в некоторых случаях переменная может указывать базовую платформу для MACHINE.

Множество BSP могут использовать общее имя KMACHINE, если они собираются с использованием одного описания BSP. Например, множество BSP на базе Corei7 использует KMACHINE=intel-corei7-64. Важно понимать, что KMACHINE служит просто для сопоставления с ядром, а MACHINE указывает тип машины на уровне BSP. Однако даже с учётом этого различия переменные могут иметь одинаковые значения (см. параграф 3.3.5. Описания BSP).

Каждое задание в стиле linux-yocto должно также указывать ветвь репозитория ядра Linux, используемую для сборки ядра, в переменной KBRANCH. Значение KBRANCH можно использовать для определения дополнительной ветви (обычно с переопределением машины), как показано ниже (из уровня meta-yocto-bsp).

```
KBRANCH_edgerouter = "standard/edgerouter"
```

Задания в стиле linux-yocto могут также определять переменные KERNEL_FEATURES и LINUX_KERNEL_TYPE.

Переменная LINUX_KERNEL_TYPE определяет тип ядра, используемый при сборке конфигурации (по умолчанию standard). Вместе с KMACHINE переменная LINUX_KERNEL_TYPE определяет аргументы поиска, используемые инструментами ядра для нахождения нужных описаний в метаданных ядра, с которыми будет выполняться сборка и настройка. Задания linux-yocto определяют типы standard, tiny и preempt-rt (см. параграф 3.3.4. Типы ядер).

В процессе сборки kern-tools выполняет поиск файла описания BSP, наиболее точно соответствующего переменным KMACHINE и LINUX_KERNEL_TYPE, переданным из задания. Инструменты используют первое найденное описание BSP, которое соответствует обеим переменным. Если описание не найдено, выдаётся предупреждение. Поиск выполняется сначала для переменной KMACHINE, а затем для LINUX_KERNEL_TYPE. Если не удаётся найти частичного совпадения, будут применяться исходные коды из KBRANCH и конфигурация, заданная в SRC_URI.

Переменную KERNEL_FEATURES можно применять для включения возможностей (фрагменты конфигурации, patch-файлы), которые не были указаны в KMACHINE и LINUX_KERNEL_TYPE. Например, для включения свойства из файла features/netfilter/netfilter.scc следует указать

```
KERNEL_FEATURES += "features/netfilter/netfilter.scc"
```

Для включения свойства cfg/sound.scc для машины qemux86 следует использовать

```
KERNEL_FEATURES_append_qemux86 = " cfg/sound.scc"
```

Значения элементов KERNEL_FEATURES зависят от их расположения внутри метаданных ядра. Приведённые здесь примеры взяты из репозитория yocto-kernel-cache, каждая из ветвей которого содержит на верхнем уровне каталоги features и cfg (см. параграф 3.3. Синтаксис метаданных ядра).

3.3. Синтаксис метаданных ядра

Метаданные ядра включают три основных типа файлов - описания scc [1], фрагменты конфигурации и patch-файлы. Файлы scc определяют переменные и включают или иным способом указывают любые из 3 упомянутых типов файлов. Файлы описания служат для объединения всех типов метаданных ядра в единое описание исходных файлов и конфигурации, требуемое для сборки ядра, адаптированного к конкретной машине.

Файлы описаний scc служат для определения двух фундаментальных типов метаданных ядра:

- возможности;
- пакеты поддержки плат (BSP).

Возможности объединяют исходные файлы в форме исправлений (patch) и фрагментов конфигурации в модуль для многократного использования. Возможности можно применять для реализации концептуально разделённых описаний метаданных ядра, таких как фрагменты конфигурации, простые исправления, сложные функции и типы ядра. Тип ядра определяет базовые функции ядра и правила их применения в BSP.

BSP определяют зависящие от оборудования возможности и объединяют их с типом ядра для создания окончательного описания того, что нужно собрать.

Хотя синтаксис метаданных ядра не требует логического разделения фрагментов конфигурации, исправлений, свойств и типов ядра, такое разделение является хорошим тоном. Ниже приведена рекомендуемая иерархия метаданных.

```
base/
   bsp/
   cfg/
   features/
   type/
   patches/
```

Каталог bsp содержит описания BSP, а в остальных каталогах описаны свойства (функции). Отделение bsp от остальной структуры способствует осмыслению предполагаемого использования. Приведённые ниже рекомендации помогут разместить файлы scc в этой структуре.

- Файлы, содержащие только фрагменты конфигурации, следует помещать в каталог cfg.
- Файлы, содержащие лишь исправления исходного кода, следует помещать в каталог patches.
- Файлы, описывающие важные свойства (major feature), зачастую объединяя исходные файлы и конфигурации, следует помещать в каталог features.
- Файлы, объединяющие не относящуюся к оборудованию конфигурацию и исправления для задания политики ядра или его типа, которые могут применяться разными BSP, следует помещать в каталог ktypes.

Отмеченные различия могут оказаться размытыми (особенно функции вне основного дерева, которые со временем попадают в него). Кроме того, следует помнить что размещение файлов является чисто логической организацией и не влияет на функциональность метаданных ядра, поскольку все каталоги cfg, features, patches и ktypes содержат те или иные свойства.

Пути в файлах метаданных ядра указываются относительно каталога base, который указан в переменной FILESEXTRAPATHS (если метаданные созданы в recipe-space) или является верхним уровнем yocto-kernel-cache, если метаданные созданы вне recipe-space (см. параграф 3.4.2. Метаданные вне recipe-space).

3.3.1. Конфигурация

Простейшим элементом метаданных ядра являются параметры конфигурации. Это свойство включает один или несколько параметров конфигурации ядра в файлах фрагментов конфигурации (.cfg) и файлах .scc с описаниями.

В качестве примера рассмотрим фрагмент симметричной многопроцессорной обработки (SMP) в ядре linux-yocto-4.12, определённый за пределами пространства задания (т. eyocto-kernel-cache). Эти метаданные включают 2 файла - smp.scc и smp.cfg. Эти файлы можно найти в каталоге cfg ветви yocto-4.12 репозитория Git yocto-kernel-cache.

```
cfg/smp.scc:
   define KFEATURE_DESCRIPTION "Enable SMP for 32 bit builds"
   define KFEATURE_COMPATIBILITY all

   kconf hardware smp.cfg

cfg/smp.cfg:
```

```
CONFIG_SMP=y
CONFIG_SCHED_SMT=y
# Increase default NR_CPUS from 8 to 64 so that platform with
# more than 8 processors can be all activated at boot time
CONFIG_NR_CPUS=64
# The following is needed when setting NR_CPUS to something
# greater than 8 on x86 architectures, it should be automatically
# disregarded by Kconfig when using a different arch
CONFIG_X86_BIGSMP=y
```

Базовая информация о фрагментах конфигурации приведена в параграфе 2.6.3. Создание фрагментов конфигурации.

В файле smp.scc оператор KFEATURE_DESCRIPTION задаёт краткое описание фрагмента, используемое инструментами верхнего уровня. Команда kconf в этом файле включает реальный фрагмент конфигурации, а ключевое слово hardware указывает, что фрагмент включает оборудование в отличие от базовой политики, использующей ключевое слово non-hardware. Это различие помогает инструментам проверки конфигурации, которые будут выдавать предупреждение, если аппаратный фрагмент переопределяет правило, установленное фрагментом non-hardware.

Файл описания может включать множество операторов kconf (по одному на фрагмент).

Как указано в параграфе 2.6.4. Проверка корректности конфигурации, вы можете использовать приведённую ниже команду BitBake для аудита вашей конфигурации.

```
$ bitbake linux-yocto -c kernel_configcheck -f
```

3.3.2. Исправления

Описания исправления (patch) очень похожи на описания фрагментов конфигурации, представленные в предыдущем параграфе. Однако вместо файлов .cfg они работают с файлами .patch.

Типичное исправление включает файл описания и сам patch-файл. Рассмотрим в качестве примера исправления сборки, используемые в ядре linux-yocto-4.12 и определённые за пределами пространства задания (т. е. yocto-kernel-cache). Эти метаданные включают несколько файлов - build.scc и набор файлов *.patch, которые можно посмотреть в каталоге patches/build ветви yocto-4.12 в репозитории Git yocto-kernel-cache.

Ниже приведено содержимое файлов build.scc и modpost-mask-trivial-warnings.patch.

```
patches/build/build.scc:
   patch arm-serialize-build-targets.patch
   patch powerpc-serialize-image-targets.patch
   patch kbuild-exclude-meta-directory-from-distclean-processi.patch

   # applied by kgit
   # patch kbuild-add-meta-files-to-the-ignore-li.patch

   patch modpost-mask-trivial-warnings.patch
   patch menuconfig-check-lxdiaglog.sh-Allow-specification-of.patch

patches/build/modpost-mask-trivial-warnings.patch:
   From bd48931bc142bdd104668f3a062a1f22600aae61 Mon Sep 17 00:00:00 2001
   From: Paul Gortmaker <paul.gortmaker@windriver.com>
   Date: Sun, 25 Jan 2009 17:58:09 -0500
   Subject: [PATCH] modpost: mask trivial warnings

   Newer HOSTCC will complain about various stdio fcns because
                    .
                    .
                    .
           char *dump_write = NULL, *files_source = NULL;
           int opt;
   --
   2.10.1
```

Файл описания может включать множество операторов patch, каждый из которых относится к одному файлу исправлений .patch. В приведённом примере файл build.scc содержит 5 операторов patch для пяти файлов исправления в каталоге.

Вы можете создавать файлы .patch с помощью команды diff -Nurp или git format-patch. Описание работы с файлами исправлений приведено в параграфах 2.4. Использование devtool для применения изменений к ядру и 2.5. Традиционные методы внесения изменений в ядро.

3.3.3. Свойства

Свойства являются комплексным типом метаданных ядра и содержат фрагменты конфигурации и исправления, а также могут включать другие файлы описания свойств. В качестве примера рассмотрим приведённый ниже файл.

```
features/myfeature.scc
   define KFEATURE_DESCRIPTION "Enable myfeature"

   patch 0001-myfeature-core.patch
   patch 0002-myfeature-interface.patch

   include cfg/myfeature_dependency.scc
   kconf non-hardware myfeature.cfg
```

Здесь показано использование команд patch и kconf, а также включение дополнительного файла описания возможностей с помощью команды include.

Обычно свойства менее детализированы по сравнению с фрагментами конфигурации и с большей вероятностью, чем фрагменты и исправления, описывают то, что вы захотите задать в переменной KERNEL_FEATURES задания для ядра Linux (см. параграф 3.2. Использование метаданных ядра в задании).

3.3.4. Типы ядер

Тип ядра определяет политику верхнего уровня путём объединения не связанных с оборудованием фрагментов конфигурации с исправлениями (patch), которые следует применить для сборки ядра Linux с определёнными свойствами (например, для работы в реальном масштабе времени). Синтаксически типы ядра не отличаются от свойств, описанных в параграфе 3.3.3. Свойства. Переменная LINUX_KERNEL_TYPE в задании для ядра указывает тип ядра. Например, в задании linux-yocto_4.12.bb из каталога poky/meta/recipes-kernel/linux директива require [9] включает файл poky/meta/recipes-kernel/linux/linux-yocto.inc, который задаёт используемый по умолчанию тип ядра.

```
LINUX_KERNEL_TYPE ??= "standard"
```

Другим примером будет ядро для работы в реальном масштабе времени (linux-yocto-rt_4.12.bb), задание для которого непосредственно устанавливает тип, как показано ниже.

```
LINUX_KERNEL_TYPE = "preempt-rt"
```

Задания для ядра можно найти в каталоге meta/recipes-kernel/linux каталога исходных кодов (например, poky/meta/recipes-kernel/linux/linux-yocto_4.12.bb). Работа с метаданными в заданиях описана в параграфе 3.2. Использование метаданных ядра в задании.

Для ядер Linux Yocto поддерживается три типа - standard, tiny и preempt-rt.

standard
: Включает базовые правила заданий для ядра YP linux-yocto, включающие наряду с прочим файловые системы, сетевые опции, базовые функции ядра, а также опции отладки и трассировки.

preempt-rt
: Задаёт использование правок PREEMPT_RT и параметров конфигурации, требуемых для работы ядра в реальном масштабе времени. Этот тип наследует свойства типа standard.

tiny
: Задаёт минимальную конфигурацию для создания очень компактного ядра Linux. Тип tiny не зависит от конфигурации standard. В настоящее время тип tiny не вносит изменений в исходный код, но в будущем это возможно.

Для любого типа ядра метаданные определяются в файле .scc (например, standard.scc). Ниже приведена часть файла standard.scc file из каталога ktypes/standard в репозитории Git yocto-kernel-cache.

```
# Include this kernel type fragment to get the standard features and
# configuration values.

# Note: if only the features are desired, but not the configuration
#       then this should be included as:
#             include ktypes/standard/standard.scc nocfg
#       if no chained configuration is desired, include it as:
#             include ktypes/standard/standard.scc nocfg inherit


include ktypes/base/base.scc
branch standard

kconf non-hardware standard.cfg

include features/kgdb/kgdb.scc
            .
            .
            .

include cfg/net/ip6_nf.scc
include cfg/net/bridge.scc

include cfg/systemd.scc

include features/rfkill/rfkill.scc
```

Как и другие файлы .scc, определение типа ядра может включать другие файлы .scc с помощью команд include. Эти определения могут напрямую «затягивать» фрагменты конфигурации и patch-файлы с помощью команд kconf и patch.

Не обязательно создавать файл .scc для типа ядра. Файл BSP может неявно указывать тип ядра с помощью строки KTYPE myktype (см. параграф 3.3.5. Описания BSP).

3.3.5. Описания BSP

Описания BSP (файлы *.scc) комбинируют типы ядер с аппаратно-зависимыми функциями. Связанные с оборудованием метаданные обычно определяются независимо на уровне BSP, а затем объединяются с каждым поддерживаемым типом ядра.

Для BSP, поддерживаемых в YP, файлы описаний BSP размещаются в каталоге bsp репозитория yocto-kernel-cache (раздел Yocto Linux Kernel по ссылке Yocto Project Source Repositories).

Далее приведен обзор структуры описаний BSP и концепций агрегирования, а также пример использования BSP в YP (BeagleBone Board). Полная информация об иерархии файлов уровня BSP дана в [5].

3.3.5.1. Обзор

Рассмотрим файлы описания корневого уровня BSP для платы BeagleBone. Структура и имена файлов соответствуют упомянутым выше рекомендациям. В соответствии с соглашением об именовании имя определяется как

```
bsp_root_name-kernel_type.scc
```

Ниже указаны два примера имён файлов корневого уровня BSP для BeagleBone Board BSP, поддерживаемых YP.

```
beaglebone-standard.scc
beaglebone-preempt-rt.scc
```

Каждый из файлов использует корневое имя BSP (beaglebone), за которым следует тип ядра.

Рассмотрим файл beaglebone-standard.scc

```
define KMACHINE beaglebone
define KTYPE standard
define KARCH arm

include ktypes/standard/standard.scc
branch beaglebone

include beaglebone.scc

# default policy for standard kernels
include features/latencytop/latencytop.scc
include features/profiling/profiling.scc
```

Каждому файлу описания верхнего уровня BSP следует определять переменные KMACHINE, KTYPE и KARCH, которые позволяют системе сборки OE идентифицировать описание как соответствующее критериям, установленным заданием. В этом примере поддерживается машина beaglebone с ядром standard для архитектуры arm.

Следует принимать во внимание отсутствие жёсткой привязки переменной KTYPE к файлу описания типа ядра. Если тип ядра не указан в метаданных ядра, как описано здесь, нужно убедиться в совпадении значений переменной LINUX_KERNEL_TYPE в задании для ядра и переменной KTYPE в описании BSP.

Чтобы отделить политику ядра от аппаратной конфигурации, указывается тип ядра (ktype), такой как standard. В предыдущем примере это сделано с помощью строки, приведённой ниже

```
include ktypes/standard/standard.scc
```

Файл объединяет все фрагменты конфигурации, patch-файлы и возможности, которые определяют вашу стандартную политику ядра (см. параграф 3.3.4. Типы ядер).

Для объединения базовых конфигураций и возможностей, относящихся к ядру для mybsp, служит строка

```
include mybsp.scc
```

В примере BeagleBone используется строка

```
include beaglebone.scc
```

Информация о разбиении полного файла .config на фрагменты конфигурации приведена в параграфе 2.6.3. Создание фрагментов конфигурации.

Если конфигурация, связанная с оборудованием, отсутствует в файле *.scc, её можно включить с помощью строки

```
kconf hardware mybsp-extra.cfg
```

Пример BeagleBone не включает эти типы конфигурации, однако для 32-битовой платы Malta (mti-malta32)они включены в файл mti-malta32-le-standard.scc, приведённый ниже.

```
define KMACHINE mti-malta32-le
define KMACHINE qemumipsel
define KTYPE standard
define KARCH mips

include ktypes/standard/standard.scc
branch mti-malta32

include mti-malta32.scc
kconf hardware mti-malta32-le.cfg
```

3.3.5.2. Пример

Реальные примеры в основном сложнее. Подобно другим файлам .scc, описания BSP могут объединять возможности. Рассмотрим определение Minnow BSP из ветви linux-yocto-4.4 в yocto-kernel-cache (yocto-kernel-cache/bsp/minnow/minnow.scc)

Хотя Minnow Board BSP не применяется, метаданные сохранены и будут служить здесь в качестве примера.

```
include cfg/x86.scc
include features/eg20t/eg20t.scc
include cfg/dmaengine.scc
include features/power/intel.scc
include cfg/efi.scc
include features/usb/ehci-hcd.scc
include features/usb/ohci-hcd.scc
include features/usb/usb-gadgets.scc
include features/usb/touchscreen-composite.scc
include cfg/timer/hpet.scc
include features/leds/leds.scc
include features/spi/spidev.scc
include features/i2c/i2cdev.scc
include features/mei/mei-txe.scc

# Earlyprintk and port debug requires 8250
kconf hardware cfg/8250.cfg

kconf hardware minnow.cfg
```

```
kconf hardware minnow-dev.cfg
```

Файл описания minnow.scc включает фрагмент аппаратной конфигурации (minnow.cfg) для Minnow BSP, а также несколько фрагментов конфигурации общего назначения и возможности, включающее оборудование на плате. Этот файл описания minnow.scc включается в каждый из трёх файлов описаний minnow для поддерживаемых типов ядра (standard, preempt-rt, tiny). Рассмотрим описание minnow для ядра типа standard (файл minnow-standard.scc).

```
define KMACHINE minnow
define KTYPE standard
define KARCH i386

include ktypes/standard

include minnow.scc

# Extra minnow configs above the minimal defined in minnow.scc
include cfg/efi-ext.scc
include features/media/media-all.scc
include features/sound/snd_hda_intel.scc

# The following should really be in standard.scc
# USB live-image support
include cfg/usb-mass-storage.scc
include cfg/boot-live.scc

# Basic profiling
include features/latencytop/latencytop.scc
include features/profiling/profiling.scc

# Requested drivers that don't have an existing scc
kconf hardware minnow-drivers-extra.cfg
```

Команда include включает описание minnow.scc, которое определяет оборудование BSP, общее для всех типов ядра. Это существенно снижает дублирование.

Рассмотрим описание minnow для ядра типа "tiny (minnow-tiny.scc)

```
define KMACHINE minnow
define KTYPE tiny
define KARCH i386

include ktypes/tiny

include minnow.scc
```

Легко видеть, что это описание существенно меньше и фактически включает лишь минимальные правила, заданные типом ядра tiny и зависящую от оборудования конфигурацию, требуемую для загрузки машины, вместе с базовой функциональностью системы, определённой в файле описания minnow" description file.

Ещё раз отметим 3 важных переменных - KMACHINE, KTYPE, KARCH. Из них лишь KTYPE меняется для типа tiny.

3.4. Размещение метаданных ядра

Метаданные ядра всегда размещаются за пределами дерева кодов, указанного в задании для ядра (recipe-space), или вне задания. Выбор метаданных зависит от задачи и способа её выполнения. Синтаксис метаданных не зависит от места их определения.

Если вы не имеете опыта работы с ядром Linux и хотите лишь применить конфигурацию и возможно небольшие изменения, подготовленные другими людьми, рекомендуется использовать метод recipe-space. Он также хорошо подходит для случаев работы с неконтролируемыми исходными кодами ядра или просто не хотите поддерживать свой репозиторий Git для ядра Linux. Информация об определении метаданных в recipe-space частично приведена в параграфе 2.3. Изменение существующего задания.

Если вы активно занимаетесь разработками для ядра и уже поддерживаете свой репозиторий Git, может оказаться более удобным размещение метаданных за пределами recipe-space. Такой подход позволяет повысить эффективность интерактивной разработки для ядра Linux за пределами среды BitBake.

3.4.1. Метаданные в recipe-space

При сохранении метаданных ядра в recipe-space они размещаются в иерархии каталогов под FILESEXTRAPATHS. Для задания linux-yocto или задания, созданного на основе oe-core/meta-skeleton/recipes-kernel/linux/linux-yocto-custom.bb переменная FILESEXTRAPATHS обычно имеет значение ${THISDIR}/${PN} (см. параграф 2.3. Изменение существующего задания).

Ниже представлен пример, показывающий тривиальное дерево метаданных ядра в recipe-space внутри уровня BSP.

```
meta-my_bsp_layer/
`-- recipes-kernel
    `-- linux
        `-- linux-yocto
            |-- bsp-standard.scc
            |-- bsp.cfg
            `-- standard.cfg
```

Когда метаданные хранятся в recipe-space, нужно предпринять определённые действия по обеспечению BitBake информацией, которая нужна для извлечения метаданных. Для этого нужно указать файлы .scc в переменной SRC_URI. BitBake проанализирует их и извлечёт файлы, указанные в .scc командами include, patch или kconf. Поэтому нужно менять значение PR в задании при изменении содержимого файлов, указанных в SRC_URI.

Если описание BSP находится в recipe-space, нельзя просто изменить список файлов *.scc в операторе SRC_URI. Нужно использовать приведённую ниже форму из файла добавления для ядра.

```
SRC_URI_append_myplatform = " \
   file://myplatform;type=kmeta;destsuffix=myplatform \
   "
```

3.4.2. Метаданные вне recipe-space

При сохранении за пределами recipe-space метаданные ядра размещаются в отдельном репозитории. Система сборки OE добавляет метаданные в сборку как репозиторий type=kmeta в переменной SRC_URI. В качестве примера рассмотрим оператор SRC_URI из задания для ядра linux-yocto_4.12.bb.

```
SRC_URI = "git://git.yoctoproject.org/linux-yocto-4.12.git;name=machine;branch=${KBRANCH}; \
git://git.yoctoproject.org/yocto-kernel-cache;type=kmeta;name=meta;branch=yocto-4.12;destsuffix=${KMETA}"
```

В этом контексте ${KMETA} просто указывает каталог, в который сборщик Git помещает метаданные. Э/то не отличается от поведения, заданного оператором SRC_URI для множества репозиториев в задании (см. выше).

Можно хранить метаданные ядра в кэше ядра, который является каталогом с фрагментами конфигурации. Как и для других метаданных за пределами recipe-space, нужно просто использовать оператор SRC_URI с атрибутом type=kmeta, что делает метаданные ядра доступными при настройке конфигурации.

При изменении метаданных нужно обновлять операторы SRCREV в задании для ядра. В частности, нужно обновлять переменную SRCREV_meta в соответствии с фиксацией (commit) в используемой ветви KMETA. Изменение данных в этих ветвях без соответствующего обновления операторов SRCREV приведёт к использованию при сборке старых фиксаций.

3.5. Организация исходных кодов

Многие задания, основанные на linux-yocto-custom.bb используют исходные коды ядра Linux с единственной ветвью master. Этот тип структуры репозитория отлично подходит для разработок, поддерживающих одну машину и архитектуру. Однако при работе с разными платами или архитектурой эффективней использовать репозиторий исходных кодов с множеством ветвей. Предположим, например, что вам нужна серия исправления для загрузки одной платы. Иногда эти исправления находятся в стадии разработки или в корне ошибочны, но все же нужны для определённых плат. В таких ситуациях вы скорей всего не захотите включать эти исправления в каждую сборку ядра (т. е. в ветвь master). Именно для таких ситуация применяется ветвление в репозиториях Git для ядра Linux.

Существуют стратегии организации репозитория, максимизирующие многократное использование кода, устраняющие избыточность и логически упорядочивающие внесённые изменения. В этом параграфе представлена стратегия для нескольких случаев.

- Инкапсуляция исправлений в описание свойства и включение patch-файлов только в описания BSP соответствующих плат.
- Создание ветви для машины в вашем репозитории кодов и применение исправлений лишь к этой ветви.
- Создание ветви для возможности и при необходимости слияние этой ветви с BSP.

Выбор подхода полностью определяется вами с учётом вашей модели разработки.

3.5.1. Инкапсуляция исправлений

Если вы многократно используете patch-файлы из внешнего дерева и не создаёте своих исправлений, инкапсуляция может быть подходящим решением. В этом варианте не нужно создавать ветвей в репозитории исходных кодов. Просто берутся нужные статические patch-файлы и инкапсулируются в описание свойства, которое включается в описание BSP, как указано в параграфе 3.3.5. Описания BSP.

3.5.2. Ветви для машин

При работе с несколькими машинами и разной архитектурой или активной поддержке определённых плат более эффективно создавать в репозитории ветви для конкретных машин. Это позволяет сохранять для общего дерева кодов ветвь master, а относящиеся к конкретной машине свойства хранить в ветви для этой машины. Такой метод освобождает от необходимости постоянно встраивать ваши исправления в свойство.

При создании новой ветви можно организовать метаданные для использования ветви разными способами. В задании можно указать новую ветвь как KBRANCH для использования с платой

```
KBRANCH = "mynewbranch"
```

Другой метод использует команду branch в описании BSP.

```
mybsp.scc:
   define KMACHINE mybsp
   define KTYPE standard
   define KARCH i386
   include standard.scc

   branch mynewbranch

   include mybsp-hw.scc
```

При наличии множества ветвей можно применять структуру, аналогичную используемой в репозиториях Yocto Linux Kernel Git

```
common/kernel_type/machine
```

Если вы используете два типа ядер (например, standard и small), три машины и общий корень mydir, ветви вашего репозитория могут иметь вид

```
mydir/base
```

```
mydir/standard/base
mydir/standard/machine_a
mydir/standard/machine_b
mydir/standard/machine_c
mydir/small/base
mydir/small/machine_a
```

Такая организация позволяет ясно видеть связи между ветвями. В приведённом примере mydir/standard/machine_a включает все из mydir/base mydir/standard/base. Ветви standard и small добавляют исходные коды, относящиеся к этим ветвям и по той или иной причине не подходящие для других ветвей.

Ветвь base является деталью внутреннего управления Git в своей файловой системе, не позволяя использовать mydir/standard и mydir/standard/machine_a, поскольку это будет создавать файл и каталог с именем standard.

3.5.3. Ветви для свойств

При активной разработке новых свойств может оказаться более эффективным ветвление по свойствам, а не наборам исправлений, которые нужно регулярно обновлять. Инструменты для ядер YP Linux позволяют делать это с помощью команды git merge.

Для слияния ветви по свойству с BSP нужно поместить команду git merge после каждой команды branch.

```
mybsp.scc:
   define KMACHINE mybsp
   define KTYPE standard
   define KARCH i386
   include standard.scc

   branch mynewbranch
   git merge myfeature

   include mybsp-hw.scc
```

3.6. Команды в файлах описаний

Здесь приведен краткий обзор команд, которые можно использовать в файлах описаний (.scc)

branch [ref]
Создаёт новую ветвь (обычно ${KTYPE}), используя выбранную или указанную параметром ref ветвь.

define
Определяет переменные (такие как KMACHINE, KTYPE, KARCH, KFEATURE_DESCRIPTION).

include SCC_FILE
Включает файл SCC в текущий файл с анализом как при встраивании (inline).

kconf [hardware|non-hardware] CFG_FILE
Помещает фрагмент конфигурации в очередь для слияния в финальный файл конфигурации ядра Linux (.config).

git merge GIT_BRANCH
Сливает ветвь свойства с текущей ветвью.

patch PATCH_FILE
Применяет исправление (patch) к текущей ветви Git.

В [1] scc считается сокращением от Series Configuration Control (управление последовательностью конфигураций), но в текущей реализации это уже частично утратило смысл и файлы .scc стали файлами описаний.

Приложение A. Дополнительные вопросы

A.1. Разработка и поддержка YP Kernel

Ядра в YP (ядра Yocto Linux), как и другие ядра, имеют в своей основе исходные коды выпусков ядра Linux с сайта http://www.kernel.org. В начале цикла создания новой версии ядра Linux команда YP выбирает версию ядра Linux с учётом времени выпуска, прогнозируемого срока выпуска финальной версии и функциональных требований YP. Обычно выбирается ядро на финальной стадии разработки сообществом Linux. Иными словами, это предварительный выпуск (rc), который ещё не стал финальным. Но на заключительных этапах разработки команда знает о близости окончательно выпуска, который наступает на ранней стадии разработки YP.

Это позволяет команде YP выпускать современные версии ядер Yocto Linux, соответствующие стабильным выпускам базового ядра Linux.

Как отмечено выше, основным источником для ядер Yocto Linux являются ядра с kernel.org. В дополнение к этому ядра Yocto Linux включают набор важных основных (mainline) и не основных (когда нет вариантов) разработок, BSP и специальных возможностей. Это обеспечивает выпуск ядер YP Linux, соответствующих потребностям разработчиков встраиваемых систем для целевых платформ.

Web-интерфейс для доступа к ядрам Yocto Linux в репозиториях исходных кодов доступен по ссылке http://git.yoctoproject.org. В этом интерфейсе вы увидите слева группу репозиториев Git под рубрикой Yocto Linux Kernel, в которой доступно несколько выпусков ядра Linux Yocto.

linux-yocto-4.1
Стабильное ядро YP для использования с YP 2.0 на основе ядра Linux 4.1.

linux-yocto-4.4
Стабильное ядро YP для использования с YP 2.1 на основе ядра Linux 4.4.

linux-yocto-4.6
Временное ядро, не привязанное к выпуску YP.

linux-yocto-4.8
Стабильное ядро YP для использования с YP 2.2.

linux-yocto-4.9
Стабильное ядро YP для использования с YP 2.3 на основе ядра Linux 4.9.

linux-yocto-4.10
Стабильное ядро YP для использования с YP 2.3 на основе ядра Linux 4.10.

linux-yocto-4.12
Стабильное ядро YP для использования с YP 2.4 на основе ядра 4.12.

yocto-kernel-cache
Исправления (patch) и конфигурационные файлы для дерева linux-yocto (Глава 3. Работа с расширенными метаданными (yocto-kernel-cache).

linux-yocto-dev
Находящееся в разработке ядро для последнего доступного rc.

Инициатива долгосрочной поддержки LTSI[1] для ядер Yocto Linux включает:

- для YP 1.7, 1.8, 2.0 - linux-yocto-3.14;
- для YP 2.1, 2.2, 2.3 - linux-yocto-4.1;
- для YP 2.4 - linux-yocto-4.9.
- linux-yocto-4.4 является ядром LTS[2].

Как только ядро Yocto Linux выпущено официально, команда YP начинает следующий цикл разработки или цикл uprev[3], сохраняя поддержку выпущенного ядра. Важно подчеркнуть, что наиболее устойчивый и стабильный способ включения потока будущей разработки выполняется через процесс uprev. Перенос сотен исправлений и второстепенных возможностей из разных версий ядра не обеспечивает устойчивость и может стать угрозой качеству.

В цикле uprev команда YP использует текущий анализ разработки ядра Linux, поддержки BSP и время выпуска для выбора наиболее подходящей версии с kernel.org, на основе которой будет создаваться последующая версия ядра Yocto Linux. Команда постоянно отслеживает разработку ядра сообществом Linux для фиксации интересных возможностей. Рассматривается также перенос существенных возможностей в прежние версии, если это обеспечит преимущество. Запросы пользователей и сообщества также могут инициировать перенос возможностей в прежние версии или добавление функциональности в базовое ядро YP в цикле uprev.

Вообще говоря, каждое новое ядро вносит новые функции и добавляет новые ошибки. Эти свойства разработки ядер Linux контролируются стратегией разработки ядер команды YP. Политика команды заключается в том, чтобы не переносить маловажные свойства в прошлые выпуски ядер Yocto Linux. Рассматривается лишь возможность переноса существенных технологических прорывов и только после их полного анализа. Причина этого заключается в том, что перенос малозначимых возможностей в прошлые выпуски может приводить к несоответствиям, несовместимости и коварным ошибкам.

Описанная здесь политика позволяет сделать ядро Yocto Linux передовым и стабильным, объединяющим имеющиеся возможности ядра Linux с важными новыми функциями. Перенос перспективных функций ядра Linux в ядро Yocto Linux доступный в YP, можно рассматривать как микро-uprev. Множество таких переносов создает версию ядра Yocto Linux, сочетающую важные новые функции с разработками BSP. Такое ядро Yocto Linux даёт представление о новых возможностях и позволяет выполнять целевые тесты, предотвращающие сюрпризы при выборе следующего основного uprev. Качество таких ядер Yocto Linux повышается и они служат основой для разработок.

A.2. Архитектура Yocto Linux Kernel и стратегия ветвления

Как было отмечено выше, основной целью YP является предоставление разработчикам ядра с чёткой и непрерывной историей, видимой пользователям. Используемая архитектура и механизмы (в частности, стратегия ветвления) обеспечивают достижение этой цели аналогично разработке ядра Linux сообществом kernel.org.

Ядро Yocto Linux можно рассматривать как базовое ядро Linux с добавлением логически структурированных функций, которые помечены и организованы с помощью стратегии ветвления, реализованной командой YP на основе SCM[4] Git.

- Git является очевидным выбором SCM с учётом организационных и структурных потребностей Yocto Linux, описанных здесь. Git не только служит SCM для разработки ядра Linux на kernel.org, но и продолжает набирать популярность и поддержку в разных проектах, системах и методах управления.
- Документация для Git доступна на сайте http://git-scm.com/documentation. Вводная информация о Git в контексте YP представлена в разделе Git [7], где приведен обзор и описан минимальный набор команд Git для работы с YP.

Используя теги и ветви Git, команда YP организует ветвление ядра в точках, где функциональность перестаёт быть общей и должна быть отделена. Например, связанные с платой несовместимости могут требовать других функций с их выделением в особую ветвь. Аналогичный подход применяется и для ветвления по свойствам ядра.

Эта древовидная архитектура приводит к созданию структуры, организованной по конкретной функциональности, типам или подтипам ядер. В результате пользователь может видеть добавленные свойства и связанные с ними фиксации (commit). Кроме того, пользователь может видеть историю изменения базового ядра Linux.

Другим следствием этой стратегии является отсутствие необходимости дважды сохранять одно свойство в дереве. Вместо этого сохраняется уникальное различие, требуемое для применения свойства к нужному типу ядра.

Команда YP старается размещать свойства в дереве так, чтобы их можно было применять для всех возможных плат и типов ядра. Однако в процессе разработки или при объединении крупных функций (свойств) такой подход возможен не всегда. В таких случаях для объединения свойств применяется ветвление.

Связанные с BSP дополнения кода обрабатываются похожим способом. Некоторые BSP имеют смысл лишь для отдельных типов ядра и для таких типов создаются ветви для всех BSP, поддерживаемых этим типом ядра. С точки зрения инструментов, создающих ветвь BSP, нет разницы между свойством и BSP. Поэтому для BSP применяется та

[1]Long Term Support Initiative.
[2]Long Term Support - долгосрочная поддержка.
[3]Upward revision - пересмотр вверх.
[4]Source Code Manager - менеджер исходных кодов.

же стратегия ветвления и вместо сохранения BSP дважды записывается лишь уникальное различие между BSP для множества поддерживаемых ядер.

Хотя эта стратегия может приводить к росту числа ветвей дерева, важно понимать, что с точки зрения разработчика есть линейный путь от базового ядра kernel.org через выбор группы возможностей к связанным с BSP фиксациям (commit). Этим путём для разработчика является ветвь master в терминах Git. Разработчику не нужно даже знать о наличии иных ветвей. Эти ветви имеют ценность для тех, кто захочет их исследовать. Например, сравнение двух BSP на уровне фиксации или построчных различий в коде становится тривиальной операцией.

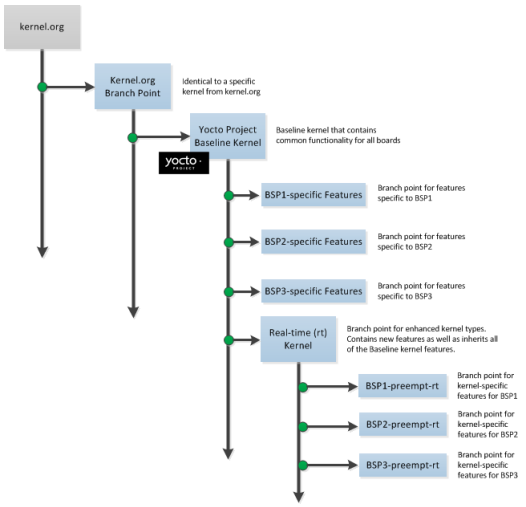


Рисунок 1. Ветвление в ядре Yocto Linux.

На рисунке Kernel.org Branch Point указывает конкретную точку (выпуск ядра Linux), где создаётся ядро Yocto Linux. От этой точки в дереве организуются и помечаются свойства и различия.

Ветвь Yocto Project Baseline Kernel содержит общие для всех типов ядра и BSP функции, организованные в форме дерева. Такое размещение основных свойств в дереве означает отсутствие дублирования свойств в разных ветвях.

Ветви Yocto Project Baseline Kernel представляют конкретную функциональность для отдельных BSP, а также ядра для работы в реальном масштабе времени. На рисунке показаны ветви для трёх BSP и ветвь real-time, каждая из которых представляет уникальную функциональность.

В показанной структуре ветвь Real-time (rt) Kernel имеет свойства, которые относятся ко всем ядрам real-time Yocto Linux, и дополнительные ветви для отдельных BSP в таких ядрах.

Древовидная структура представляет разработчику чёткую последовательность маркеров (ветвей) для всех практических задач.

Рисунок служит лишь для иллюстрации и на нем показаны далеко не все ядра Yocto Linux. Следует также учитывать. Что эта структура представляет репозитории исходных кодов YP [7], которые загружаются при сборке или организуются сборочным хостом до начала сборки путём клонирования репозитория Git или загрузки и распаковки архива.

Работа с ядром, структурированным как дерево, подтверждена практикой. В частности, представленное ядро следует считать «восходящим источником» и рассматривать как последовательность документированных изменений (фиксаций). Эти изменения представляют разработку и стабилизацию, выполненную командой YP.

Поскольку изменения фиксируются лишь для значимых точек выпуска в жизненном цикле продукции, разработчики могут использовать ветвь, созданную из последней соответствующей фиксации (commit) в представленном ядре YP Linux. Как отмечено выше, структура понятна разработчикам, поскольку ядро дерева сохраняет состояние после клонирования и сборки.

A.3. Иерархия файлов для сборки ядра

Восходящее хранилище доступного исходного кода ядра и представление кода в вашей системе разработки — это разные сущности. Концептуально можно представлять репозиторий исходных кодов ядра как все исходные файлы, требуемые для всех поддерживаемых ядер Yocto Linux. Как разработчику, вам нужны исходные коды для ядра, с которым вы работаете, поэтому они нужны на сборочном хосте. Получить исходные коды на сборочный хост можно несколькими способами.

- *Доступ с использованием devtool* из состава YP является предпочтительным методом работы с ядром (см. параграф 1.2. Рабочий процесс изменения ядра).
- *Клонированный репозиторий.* При постоянной работе с ядром имеет смысл организовать локальный репозиторий Git для ядра Yocto Linux. Информация о клонировании приведена в параграфе 2.1. Подготовка сборочного хоста для работы с ядром.
- *Временные исходные файлы из сборки.* Если нужно просто применить некоторые изменения к ядру с использованием обычных процессов BitBake (т. е. без devtool), можно работать с временными исходными файлами, которые были загружены и использовались при сборке ядра.

Временные файлы исходных кодов, полученные при сборке ядра с помощью BitBake, организованы в определённую иерархию. При сборке ядра на вашей системе все нужные файлы извлекаются из репозиториев, указанных переменной SRC_URI [1], и собираются во временной рабочей области для создания ядра. Таким образом, организуется локальное дерево исходных кодов для создаваемого образа ядра.

На рисунке 2 показана временная файловая структура, создаваемая на сборочном хосте при создании ядра с использованием Bitbake. Этот сборочный каталог содержит все файлы, нужные для сборки.

Дополнительная информация об архитектуре ядра YP и стратегии ветвления приведена в приложении A.2. Архитектура Yocto Linux Kernel и стратегия ветвления, а в параграфах 2.4. Использование devtool для применения изменений к ядру и 2.5. Традиционные методы внесения изменений в ядро приведены примеры внесения изменений в ядро.

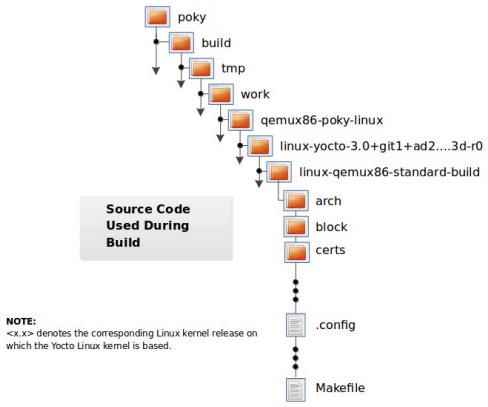


Рисунок 2. Структура файлов временного репозитория.

A.4. Указание свойств ядра для фазы проверки конфигурации

В этом параграфе частично описана фаза аудита ядра, которую большинство разработчиков могут пропустить. Общая информация о настройке конфигурации ядра приведена в параграфе 2.6. Настройка конфигурации ядра.

На этапе аудита конфигурации содержимое файла .config сравнивается с фрагментами, заданными системой (файлы фрагментов конфигурации, фрагменты дистрибутива, заданные пользователем элементы конфигурации). Независимо от происхождения этих фрагментов система сборки OE выдаёт предупреждения при отсутствии заданной фрагментом опции в окончательной конфигурации ядра.

По умолчанию система предупреждает лишь об отсутствии «аппаратных» опций, поскольку это может приводить к отказам при загрузке образа или говорить о недоступности важных компонент оборудования.

Для разделения «аппаратных» и «неаппаратных» опций метаданные ядра в yocto-kernel-cache включают файлы с классификацией отдельных опций или их групп. Рассмотрим в качестве примера yocto-kernel-cache с файлами

```
yocto-kernel-cache/features/drm-psb/hardware.cfg
yocto-kernel-cache/features/kgdb/hardware.cfg
yocto-kernel-cache/ktypes/base/hardware.cfg
yocto-kernel-cache/bsp/mti-malta32/hardware.cfg
yocto-kernel-cache/bsp/fsl-mpc8315e-rdb/hardware.cfg
yocto-kernel-cache/bsp/qemu-ppc32/hardware.cfg
yocto-kernel-cache/bsp/qemuarma9/hardware.cfg
yocto-kernel-cache/bsp/mti-malta64/hardware.cfg
yocto-kernel-cache/bsp/arm-versatile-926ejs/hardware.cfg
yocto-kernel-cache/bsp/common-pc/hardware.cfg
yocto-kernel-cache/bsp/common-pc-64/hardware.cfg
yocto-kernel-cache/features/rfkill/non-hardware.cfg
yocto-kernel-cache/ktypes/base/non-hardware.cfg
yocto-kernel-cache/features/aufs/non-hardware.kcf
yocto-kernel-cache/features/ocf/non-hardware.kcf
yocto-kernel-cache/ktypes/base/non-hardware.kcf
yocto-kernel-cache/ktypes/base/hardware.kcf
yocto-kernel-cache/bsp/qemu-ppc32/hardware.kcf
```

Ниже приведены краткие разъяснения для некоторых из этих файлов.

- *hardware.kcf* задаёт список файлов Kconfig, включающих только аппаратные опции.
- *non-hardware.kcf* задаёт список файлов Kconfig, включающих только неаппаратные опции.
- *hardware.cfg* задаёт список опций ядра CONFIG_ options, которые являются аппаратными независимо от указания в той или иной категории файлов Kconfig (например, hardware.kcf или non-hardware.kcf).
- *non-hardware.cfg* задаёт список опций ядра CONFIG_ options, которые не являются аппаратными независимо от указания в той или иной категории файлов Kconfig (например, hardware.kcf или non-hardware.kcf).

Ниже приведен пример использования файла kernel-cache/bsp/mti-malta32/hardware.cfg:

```
CONFIG_SERIAL_8250
CONFIG_SERIAL_8250_CONSOLE
CONFIG_SERIAL_8250_NR_UARTS
CONFIG_SERIAL_8250_PCI
CONFIG_SERIAL_CORE
CONFIG_SERIAL_CORE_CONSOLE
CONFIG_VGA_ARB
```

При аудите конфигурации ядра упомянутые файлы (они должны иметь указанные здесь имена) будут автоматически найдены и использованы в качестве входных данных для сравнения с окончательным файлом .config.

В пользовательском репозитории метаданных ядра или recipe-space могут применяться такие же имена файлов для классификации опций как аппаратных или неаппаратных, чтобы система сборки OE не выдавала предупреждений или ошибок при отсутствии опции в файле .config.

Приложение B. Поддержка ядра

B.1. Организация дерева

Здесь описано создание репозиториев исходного кода ядер YP, выполняемое командой YP. Эти репозитории можно найти в рубрике Yocto Linux Kernel на сайте http://git.yoctoproject.org и они включаются в выпуски YP. Репозитории создаются путём компиляции и выполнения набора описаний свойств для каждого BSP и свойства в проекте. Эти описания свойств указывают все требуемые исправления (patch), конфигурации, ветви, теги и свойства для ядра Yocto Linux. Таким образом, создаётся репозиторий (дерево) ядра YP Linux и сопровождающие его метаданные в yocto-kernel-cache.

Наличие этих репозиториев позволяет получить доступ и клонировать нужный репозиторий ядра YP Linux, а затем использовать его для сборки своих конфигураций.

Файлы, используемые для описания всех действительных свойств и BSP ядра YP Linux в любом клоне репозитория YP Linux и деревьях Git yocto-kernel-cache. Например, приведённые ниже команды будут клонировать базовый репозиторий ядра YP Linux, созданный на основе ядра версии 4.12 и yocto-kernel-cache, где содержатся метаданные.

```
$ git clone git://git.yoctoproject.org/linux-yocto-4.12
$ git clone git://git.yoctoproject.org/linux-kernel-cache
```

Дополнительная информация о создании локального репозитория Git с файлами ядра YP Linux приведена в параграфе 2.1. Подготовка сборочного хоста для работы с ядром.

После клонирования репозитория Git и кэша метаданных на локальную машину можно просмотреть все доступные ветви репозитория с помощью команды

```
$ git branch -a
```

Выбор ветви позволяет работать с определенным ядром Yocto Linux. Например, приведённые ниже команды выбирают ветвь standard/beagleboard из репозитория Yocto Linux и ветвь yocto-4.12 из yocto-kernel-cache.

```
$ cd ~/linux-yocto-4.12
$ git checkout -b my-kernel-4.12 remotes/origin/standard/beagleboard
$ cd ~/linux-kernel-cache
$ git checkout -b my-4.12-metadata remotes/origin/yocto-4.12
```

Ветви репозитория yocto-kernel-cache соответствуют версиям Yocto Linux (например, yocto-4.12, yocto-4.10, yocto-4.9).

После выбора нужно ветви и переключения на неё можно видеть «моментальный снимок» (snapshot) всех файлов исходного кода ядра, используемых для сборки ядра Yocto Linux под конкретную плату. Для просмотра свойств и конфигураций определённого ядра Yocto Linux нужно обратиться к репозиторию yocto-kernel-cache. Как отмечено выше, ветви в yocto-kernel-cache соответствуют версиям ядра Yocto Linux (например, yocto-4.12). Ветви содержат описания в файлах .scc и .cfg.

Следует учитывать. Что просмотр локального репозитория yocto-kernel-cache в части описаний возможностей и patch-файлов не является эффективным способом определения содержимого конкретной ветви. Вместо этого следует использовать Git напрямую для поиска изменений. Это обеспечивает эффективный и гибкий способ просмотра изменений в ядре.

Базовая реконструкция всего дерева ядра предпринимается командой YP только в активном цикле разработки. Создание клона репозитория ядра делает его доступным для сборки и внесения изменений.

Ниже перечислены этапы создания командой YP репозитория (дерева) исходных кодов ядра для сайта http://git.yoctoproject.org с учётом добавления новых свойств верхнего уровня и BSP. Приведённые действия обеспечивают метаданные и создание дерева с новыми возможностями, исправлениями и BSP.

1. *Передача свойства системе сборки OE.* Свойство верхнего уровня для ядра передаётся системе сборки. Обычно этим свойством является BSP для определённого типа ядра.
2. *Размещение свойства.* Файл с описанием свойства верхнего уровня отыскивается в одном из каталогов:
   - каталоги кэширования в дереве ядра, расположенные в репозитории yocto-kernel-cache под рубрикой Yocto Linux Kernel в Yocto Project Source Repositories;
   - области, указанные операторами SRC_URI в заданиях ядра.

   Для типичной сборки целью поиска является описание свойства в файле .scc имя которого соответствует приведённому ниже формату (например, beaglebone-standard.scc или beaglebone-preempt-rt.scc).

   ```
   bsp_root_name-kernel_type.scc
   ```
3. *Извлечение свойства.* Найденное описание свойства извлекается в простой сценарий действий или в имеющийся эквивалентный сценарий, являющийся частью распространяемого ядра.
4. *Добавление дополнительных свойств.* С описанию свойства верхнего уровня добавляются дополнительные возможности, которые могут определяться из переменной KERNEL_FEATURES в задании.
5. *Нахождение, извлечение и добавление каждого свойства.* Каждое дополнительное свойство отыскивается, извлекается и добавляется в сценарий, как описано в п. 3.
6. *Выполнение сценария* для создания файлов .scc и .cfg в нужных каталогах репозитория yocto-kernel-cache. Эти файлы являются описаниями всех ветвей, тегов, исправлений и конфигураций, которые нужно применить к базовому репозиторию Git для завершения создания ветви (сборки) для нового BSP или свойства.
7. *Клонирование базового репозитория* и применение действий, указанных в каталогах yocto-kernel-cache, к дереву исходных кодов.
8. *Очистка.* Репозитории Git остаются с выбранными ветвями и выполняются все требуемые ветвления, исправления и пометки.

Дерево исходных кодов и кэш готовы для передачи разработчикам, которые будут его клонировать, настраивать и собирать ядра YP для конкретных целевых платформ.

- Созданный репозиторий yocto-kernel-cache добавляется к ядру, распространяемому с выпуском YP. Все дополнения и данные конфигурации применяются в конце имеющейся ветви. Полный вариант репозитория, доступный по ссылке http://git.yoctoproject.org, включает все поддерживаемые платы и конфигурации.
- Используемый командой YP метод достаточно гибок и позволяет аккуратно смешивать неизменяемую историю с дополнительными правками, относящимися к конкретному развёртыванию. Все дополнения к ядру становятся частями ветвей.
- Генерируется полное дерево для http://git.yoctoproject.org с помощью перечисленных выше шагов для всех действительных BSP. Конечным результатом является разветвлённое дерево с чистой историей, которое образует ядро для данного выпуска. Сценарий, с помощью которого выполняются эти операции (kgit-scc), доступен в репозитории yocto-kernel-tools.
- Этапы создания полного дерева ядра совпадают с используемыми BitBake этапами сборки образа ядра.

B.2. Стратегия сборки

После клонирования репозитория ядра Yocto Linux и yocto-kernel-cache на свой хост вы можете перейти к компиляции ядра и сборке образа. Однако нужно выполнить ряд предварительных условий, которые проверяются при сборке.

- Переменная SRC_URI должна указывать репозиторий ядра.
- Ветвь сборки BSP с метаданными должна присутствовать в репозитории yocto-kernel-cache. Эта ветвь базируется на версии ядра Yocto Linux и включает конфигурации и свойства, собранные в каталоге yocto-kernel-cache/bsp. Например, свойства и конфигурации для платы BeagleBone с ядром linux-yocto_4.12 предполагаются в каталоге

  ```
  yocto-kernel-cache/bsp/beaglebone
  ```

  Здесь выбрана ветвь yocto-4.12 репозитория yocto-kernel-cache.

Система сборки OE проверяет выполнение этих условий до начала компиляции. Однако имеются и другие способы, такие как начальная загрузка BSP.

Перед сборкой ядра проверяется дерево и настраивается конфигурация ядра путём обработки фрагментов, заданных описаниями свойств в файлах .scc. По мере компиляции свойств связанные с ними фрагменты конфигурации ядра помечаются и записываются в каталоги в порядке компиляции. Фрагменты переносятся, обрабатываются и передаются подсистеме lkc[1] как необработанные входные данные в виде файла .config. Система lkc использует внутренние ограничения на основе зависимостей для окончательной обработки информации и создания финального файла .config, используемого при компиляции.

Используя архитектуру платы и другие относящиеся к делу значения, запускается процесс компиляции и создаётся образ ядра.

Процесс сборки создает дерево сборки, отличное от дерева локального репозитория Git. Это дерево сборки имеет имя включающее ${MACHINE} (метаданные платы - BSP) и тип ядра kernel_type (один из поддерживаемых YP типов, например, standard)

```
linux-${MACHINE}-kernel_type-build
```

Поддержка именования в дереве kernel.org обеспечивает эту функциональность по умолчанию.

Это поведение означает, что все созданные файла для определённой машины или BSP будут находиться в каталоге дерева сборки. Эти файлы включают окончательную конфигурацию (.config), все объектные файлы (.o), файлы .a и т. п. Поскольку каждая машина или BSP использует свой каталог сборки [1] в отдельной ветви репозитория Git, вы можете легко переходить от одной сборки к другой.

Приложение C. Вопросы разработки ядра (FAQ)

C.1. Общие вопросы

Ниже рассмотрены решения для некоторых часто возникающих задач.

C.1.1. Установка ядра в rootfs

Образ ядра (например, vmlinuz) создаётся пакетом kernel-image. Задания для образов зависят от kernel-base. Для управления установкой образа ядра в созданной корневой файловой системе служит переменная RDEPENDS_kernel-base, включающая или не включающая kernel-image.

Использование файла добавления для переопределения метаданных рассмотрено в разделе Using .bbappend Files in Your Layer [4].

C.1.2. Установка конкретных модулей ядра

Модули ядра Linux собираются отдельно. Чтобы обеспечить включение конкретного модуля в образ, нужно указать его в переменной RRECOMMENDS для соответствующей машины. Перечисленные ниже переменные полезны для установки конкретных модулей.

MACHINE_ESSENTIAL_EXTRA_RDEPENDS

MACHINE_ESSENTIAL_EXTRA_RRECOMMENDS

MACHINE_EXTRA_RDEPENDS

[1]Linux Kernel Configuration - конфигурация ядра Linux.

MACHINE_EXTRA_RRECOMMENDS

Например, приведённая ниже строка из файла qemux86.conf включит модули ab123 в образ для машины qemux86.

```
MACHINE_EXTRA_RRECOMMENDS += "kernel-module-ab123"
```

Дополнительная информация приведена в параграфе 2.10.2. Встраивание внешних модулей.

C.1.3. Изменение команды загрузки ядра

Командная строка ядра Linux обычно задаётся в конфигурации машины с помощью переменной APPEND. Например, для добавления отладочной информации можно задать

```
APPEND += "printk.time=y initcall_debug debug"
```

Литература

[1] Yocto Project Reference Manual, https://www.yoctoproject.org/docs/2.7.1/ref-manual/ref-manual.html (перевод).

[2] Menuconfig, https://en.wikipedia.org/wiki/Menuconfig.

[3] Source Repositories, http://git.yoctoproject.org/.

[4] Yocto Project Development Tasks Manual, https://www.yoctoproject.org/docs/2.7.1/dev-manual/dev-manual.html (перевод).

[5] Yocto Project Board Support Package (BSP) Developer's Guide, https://www.yoctoproject.org/docs/2.7.1/bsp-guide/bsp-guide.html (перевод).

[6] Yocto Project Quick Build, https://www.yoctoproject.org/docs/2.7.1/brief-yoctoprojectqs/brief-yoctoprojectqs.html.

[7] Yocto Project Overview and Concepts Manual, https://www.yoctoproject.org/docs/2.7.1/overview-manual/overview-manual.html (перевод).

[8] Yocto Project Application Development and the Extensible Software Development Kit (eSDK), https://www.yoctoproject.org/docs/2.7.1/sdk-manual/sdk-manual.html (перевод).

[9] BitBake User Manual, https://www.yoctoproject.org/docs/2.7.1/bitbake-user-manual/bitbake-user-manual.html (перевод).

Николай Малых

nmalykh@protokols.ru