Руководство разработчика Yocto Project BSP

Scott Rifenbark, <srifenbark@gmail.com>

Scotty's Documentation Services, INC

Этот документ основан на переводе *Yocto Project Board Support Package (BSP) Developer's Guide* для выпуска 3.0. Свежие версии оригинальных документов можно найти на странице Yocto Project documentation. Размещённые там материалы более актуальны, нежели включённые в архивы пакета Yocto Project.

Оглавление

Глава 1. Руководство разработчика пакетов поддержки плат (BSP)

BSP[1] представляет собой набор данных, определяющих поддержку конкретного устройства, группы устройств или аппаратной платформы. BSP включает информацию об аппаратных возможностях устройства и конфигурацию ядра с драйверами для поддержки аппаратных компонент, а также о программных компонентах, требуемых в дополнение к стандартным программам Linux для реализации полной функциональности платформы.

В этом руководстве приведены сведения об уровнях BSP, определена структура уровня BSP и рассмотрены вопросы настройки заданий для BSP, лицензировании и создании уровней BSP с помощью программы bitbake-layers.

1.1. Уровни BSP

Внутри уровня BSP используется определённая структура файлов. Для имён уровней BSP в составе YP обычно используется форма meta-bsp_root_name, где префикс meta- сопровождается именем машины или платформы bsp_root_name. Префикс meta- не является обязательным, но лучше использовать его, поскольку многие сценарии и инструменты YP предполагают его наличие.

Для понимания концепций уровня BSP рассмотрим BSP из состава выпусков YP, которые можно посмотреть на странице Yocto Project Source Repositories через web-интерфейс http://git.yoctoproject.org, где они размещены под заголовком Yocto Metadata Layers. Уровни, для которых не обеспечивается активная поддержка в YP, приведены в разделе Yocto Metadata Layer Archive. Каждый из этих репозиториев содержит уровень BSP, поддерживаемый в YP (например, meta-raspberrypi и meta-intel). Репозитории можно клонировать обычным способом на хост сборки, например, git clone git://git.yoctoproject.org/meta-raspberrypi. Кроме того, уровень meta-yocto-bsp включён в репозиторий poky и включает несколько BSP для Beaglebone, EdgeRouter, а также 32- и 64-битовой архитектуры IA.

Поток разработки BSP описан в разделе 1.4. Разработка BSP, а сведения о создании локального репозитория Git с исходными файлами приведены в разделе Locating Yocto Project Source Files [1].

Базовый каталог уровня (meta-bsp_root_name) является корнем этого уровня и должен быть добавлен в переменную BBLAYERS в файле conf/bblayers.conf каталога сборки, который создаётся после запуска сценария настройки среды OpenEmbedded (OE) oe-init-build-env. Добавление корневого каталога позволяет системе сборки OE распознавать уровень и собирать образ для него. Например,

[1]Board Support Package - пакет поддержки платы.

```
BBLAYERS ?= " \
  /usr/local/src/yocto/meta \
  /usr/local/src/yocto/meta-poky \
  /usr/local/src/yocto/meta-yocto-bsp \
  /usr/local/src/yocto/meta-mylayer \
  "
```

Порядок указания и приоритет BBFILE_PRIORITY для уровней в переменной BBLAYERS имеют значение. Например, при включении конфигурации машины в несколько уровней система сборки OE будет использовать последний найденный уровень с одинаковым приоритетом. Список уровней просматривается в BBLAYERS сверху вниз.

Некоторые BSP требуют или зависят от других уровней за пределами своего корня. В таких случаях нужно указать эти уровни в разделе Dependencies файла README в корневом каталоге BSP. Там же следует приводить все дополнительные инструкции по сборке для данного BSP.

Некоторые уровни служат хранилищем для других уровней BSP, такие уровни называют контейнерными. Примером контейнерного уровня может служить meta-openembedded в OE, включающий множество уровней meta-*. Подробное описание уровней дано в разделе Understanding and Creating Layers [1].

1.2. Подготовка хоста сборки для работы с уровнем BSP

В этом разделе описана подготовка хоста сборки для работы с BSP, по завершении которой можно создавать уровни в соответствии с разделом 1.8. Создание уровня BSP с помощью сценария bitbake-layers. Информация о структуре BSP дана в разделе 1.3. Пример структуры файлов.

1. *Организация среды сборки* с использованием BitBake описана в разделе Preparing the Build Host [1] для машин Linux и CROPS.
2. *Клонирование репозитория poky.* Для работы потребуется локальная копия YP Source Directory (локальный репозиторий poky). Клонирование репозитория и выбор нужной ветви описаны в разделах Cloning the poky Repository, Checking Out by Branch in Poky и Checking Out by Tag in Poky [1].
3. *Определение нужного уровня BSP.* В состав YP входит множество BSP, поддерживаемых на отдельных уровнях и в составе некоторых уровней. Для получения информации об имеющихся BSP следует обратиться к списку машин для выпуска.
4. *Клонирование уровня meta-intel BSP (необязательно).* Если устройство использует современные процессоры Intel, можно воспользоваться этим уровнем, описанным в файле README.
   a. *Выбор местоположения.* Обычно репозиторий meta-intel Git помещается в дерево источников (например, poky).
      ```
      $ cd /home/you/poky
      ```
   b. *Клонирование уровня.*
      ```
      $ git clone git://git.yoctoproject.org/meta-intel.git
      Cloning into 'meta-intel'...
      remote: Counting objects: 15585, done.
      remote: Compressing objects: 100% (5056/5056), done.
      remote: Total 15585 (delta 9123), reused 15329 (delta 8867)
      Receiving objects: 100% (15585/15585), 4.51 MiB | 3.19 MiB/s, done.
      Resolving deltas: 100% (9123/9123), done.
      Checking connectivity... done.
      ```
   c. *Выбор нужной ветви,* которая должна соответствовать используемой выпуском YP (например zeus).
      ```
      $ cd meta-intel
      $ git checkout -b zeus remotes/origin/zeus
      Branch zeus set up to track remote branch zeus from origin.
      Switched to a new branch 'zeus'
      ```
      Для просмотра доступных ветвей служит команда git branch -al (см. Checking Out By Branch in Poky [1]).
5. *Установка дополнительного уровня BSP (необязательно).* Если устройство более точно подходит к имеющемуся BSP вне уровня meta-intel, можно клонировать этот уровень BSP. Процесс идентичен клонированию meta-intel и отличается лишь именем уровня. Например, для клонирования meta-raspberrypi
   ```
   $ git clone git://git.yoctoproject.org/meta-raspberrypi
   Cloning into 'meta-raspberrypi'...
   remote: Counting objects: 4743, done.
   remote: Compressing objects: 100% (2185/2185), done.
   remote: Total 4743 (delta 2447), reused 4496 (delta 2258)
   Receiving objects: 100% (4743/4743), 1.18 MiB | 0 bytes/s, done.
   Resolving deltas: 100% (2447/2447), done.
   Checking connectivity... done.
   ```
6. *Инициализация среды сборки.* Из корня дерева кодов (poky) выполняется сценарий организации среды oe-init-build-env для установки параметров OE на сборочном хосте.
   ```
   $ source oe-init-build-env
   ```
   Сценарий, наряду с прочим, создаёт сборочный каталог внутри дерева исходных кодов и делает его текущим.

1.3. Пример структуры файлов

Единая структура каталога BSP позволяет пользователям легче разобраться с этим стандартом, а также упрощает стандартизацию программной поддержки оборудования. Описанная ниже структура включает элементы, относящиеся к системе сборки OE, но она может применяться и с другими системами, а при необходимости может быть преобразована в другой формат. Система сборки OE через стандартный механизм уровней может напрямую обращаться к описанному здесь формату. Уровень BSP содержит все аппаратные свойства в одном месте, используя стандартный формат, который будет полезен для всех применяющих платформу, независимо от системы сборки.

Спецификация BSP не включает систему сборки или иные инструменты, а сосредоточена лишь на аппаратных компонентах. Однако можно распространять свой уровень BSP вместе с системой сборки или иными инструментами. При этом важно понимать, что уровень BSP, система сборки и инструменты являются отдельными компонентами, которые могут группироваться по-разному. Следует понимать, что совместимость уровня BSP с проектом YP вносит определённые требования, перечисленные в параграфе 1.5.1. Требования к выпускаемым BSP.

Ниже представлен общий вид структуры каталогов уровня BSP, соответствующей стандарту. Однако в реальных BSP она может несколько отличаться.

```
meta-bsp_root_name/
meta-bsp_root_name/bsp_license_file
meta-bsp_root_name/README
meta-bsp_root_name/README.sources
meta-bsp_root_name/binary/bootable_images
meta-bsp_root_name/conf/layer.conf
meta-bsp_root_name/conf/machine/*.conf
meta-bsp_root_name/recipes-bsp/*
meta-bsp_root_name/recipes-core/*
meta-bsp_root_name/recipes-graphics/*
meta-bsp_root_name/recipes-kernel/linux/linux-yocto_kernel_rev.bbappend
```

Например уровень Raspberry Pi BSP из Source Respositories имеет структуру, показанную ниже.

```
meta-raspberrypi/COPYING.MIT
meta-raspberrypi/README.md
meta-raspberrypi/classes
meta-raspberrypi/classes/sdcard_image-rpi.bbclass
meta-raspberrypi/conf/
meta-raspberrypi/conf/layer.conf
meta-raspberrypi/conf/machine/
meta-raspberrypi/conf/machine/raspberrypi-cm.conf
meta-raspberrypi/conf/machine/raspberrypi-cm3.conf
meta-raspberrypi/conf/machine/raspberrypi.conf
meta-raspberrypi/conf/machine/raspberrypi0-wifi.conf
meta-raspberrypi/conf/machine/raspberrypi0.conf
meta-raspberrypi/conf/machine/raspberrypi2.conf
meta-raspberrypi/conf/machine/raspberrypi3-64.conf
meta-raspberrypi/conf/machine/raspberrypi3.conf
meta-raspberrypi/conf/machine/include
meta-raspberrypi/conf/machine/include/rpi-base.inc
meta-raspberrypi/conf/machine/include/rpi-default-providers.inc
meta-raspberrypi/conf/machine/include/rpi-default-settings.inc
meta-raspberrypi/conf/machine/include/rpi-default-versions.inc
meta-raspberrypi/conf/machine/include/tune-arm1176jzf-s.inc
meta-raspberrypi/docs
meta-raspberrypi/docs/Makefile
meta-raspberrypi/docs/conf.py
meta-raspberrypi/docs/contributing.md
meta-raspberrypi/docs/extra-apps.md
meta-raspberrypi/docs/extra-build-config.md
meta-raspberrypi/docs/index.rst
meta-raspberrypi/docs/layer-contents.md
meta-raspberrypi/docs/readme.md
meta-raspberrypi/files
meta-raspberrypi/files/custom-licenses
meta-raspberrypi/files/custom-licenses/Broadcom
meta-raspberrypi/recipes-bsp
meta-raspberrypi/recipes-bsp/bootfiles
meta-raspberrypi/recipes-bsp/bootfiles/bcm2835-bootfiles.bb
meta-raspberrypi/recipes-bsp/bootfiles/rpi-config_git.bb
meta-raspberrypi/recipes-bsp/common
meta-raspberrypi/recipes-bsp/common/firmware.inc
meta-raspberrypi/recipes-bsp/formfactor
meta-raspberrypi/recipes-bsp/formfactor/formfactor
meta-raspberrypi/recipes-bsp/formfactor/formfactor/raspberrypi
meta-raspberrypi/recipes-bsp/formfactor/formfactor/raspberrypi/machconfig
meta-raspberrypi/recipes-bsp/formfactor/formfactor_0.0.bbappend
meta-raspberrypi/recipes-bsp/rpi-u-boot-src
meta-raspberrypi/recipes-bsp/rpi-u-boot-src/files
meta-raspberrypi/recipes-bsp/rpi-u-boot-src/files/boot.cmd.in
meta-raspberrypi/recipes-bsp/rpi-u-boot-src/rpi-u-boot-scr.bb
meta-raspberrypi/recipes-bsp/u-boot
meta-raspberrypi/recipes-bsp/u-boot/u-boot
meta-raspberrypi/recipes-bsp/u-boot/u-boot/*.patch
meta-raspberrypi/recipes-bsp/u-boot/u-boot_%.bbappend
meta-raspberrypi/recipes-connectivity
meta-raspberrypi/recipes-connectivity/bluez5
meta-raspberrypi/recipes-connectivity/bluez5/bluez5
meta-raspberrypi/recipes-connectivity/bluez5/bluez5/*.patch
meta-raspberrypi/recipes-connectivity/bluez5/bluez5/BCM43430A1.hcd
meta-raspberrypi/recipes-connectivity/bluez5/bluez5brcm43438.service
meta-raspberrypi/recipes-connectivity/bluez5/bluez5_%.bbappend
meta-raspberrypi/recipes-core
meta-raspberrypi/recipes-core/images
meta-raspberrypi/recipes-core/images/rpi-basic-image.bb
meta-raspberrypi/recipes-core/images/rpi-hwup-image.bb
meta-raspberrypi/recipes-core/images/rpi-test-image.bb
meta-raspberrypi/recipes-core/packagegroups
meta-raspberrypi/recipes-core/packagegroups/packagegroup-rpi-test.bb
meta-raspberrypi/recipes-core/psplash
meta-raspberrypi/recipes-core/psplash/files
meta-raspberrypi/recipes-core/psplash/files/psplash-raspberrypi-img.h
meta-raspberrypi/recipes-core/psplash/psplash_git.bbappend
meta-raspberrypi/recipes-core/udev
meta-raspberrypi/recipes-core/udev/udev-rules-rpi
meta-raspberrypi/recipes-core/udev/udev-rules-rpi/99-com.rules
meta-raspberrypi/recipes-core/udev/udev-rules-rpi.bb
meta-raspberrypi/recipes-devtools
```

```
meta-raspberrypi/recipes-devtools/bcm2835
meta-raspberrypi/recipes-devtools/bcm2835/bcm2835_1.52.bb
meta-raspberrypi/recipes-devtools/pi-blaster
meta-raspberrypi/recipes-devtools/pi-blaster/files
meta-raspberrypi/recipes-devtools/pi-blaster/files/*.patch
meta-raspberrypi/recipes-devtools/pi-blaster/pi-blaster_git.bb
meta-raspberrypi/recipes-devtools/python
meta-raspberrypi/recipes-devtools/python/python-rtimu
meta-raspberrypi/recipes-devtools/python/python-rtimu/*.patch
meta-raspberrypi/recipes-devtools/python/python-rtimu_git.bb
meta-raspberrypi/recipes-devtools/python/python-sense-hat_2.2.0.bb
meta-raspberrypi/recipes-devtools/python/rpi-gpio
meta-raspberrypi/recipes-devtools/python/rpi-gpio/*.patch
meta-raspberrypi/recipes-devtools/python/rpi-gpio_0.6.3.bb
meta-raspberrypi/recipes-devtools/python/rpio
meta-raspberrypi/recipes-devtools/python/rpio/*.patch
meta-raspberrypi/recipes-devtools/python/rpio_0.10.0.bb
meta-raspberrypi/recipes-devtools/wiringPi
meta-raspberrypi/recipes-devtools/wiringPi/files
meta-raspberrypi/recipes-devtools/wiringPi/files/*.patch
meta-raspberrypi/recipes-devtools/wiringPi/wiringpi_git.bb
meta-raspberrypi/recipes-graphics
meta-raspberrypi/recipes-graphics/eglinfo
meta-raspberrypi/recipes-graphics/eglinfo/eglinfo-fb_%.bbappend
meta-raspberrypi/recipes-graphics/eglinfo/eglinfo-x11_%.bbappend
meta-raspberrypi/recipes-graphics/mesa
meta-raspberrypi/recipes-graphics/mesa/mesa-gl_%.bbappend
meta-raspberrypi/recipes-graphics/mesa/mesa_%.bbappend
meta-raspberrypi/recipes-graphics/userland
meta-raspberrypi/recipes-graphics/userland/userland
meta-raspberrypi/recipes-graphics/userland/userland/*.patch
meta-raspberrypi/recipes-graphics/userland/userland_git.bb
meta-raspberrypi/recipes-graphics/vc-graphics
meta-raspberrypi/recipes-graphics/vc-graphics/files
meta-raspberrypi/recipes-graphics/vc-graphics/files/egl.pc
meta-raspberrypi/recipes-graphics/vc-graphics/files/vchiq.sh
meta-raspberrypi/recipes-graphics/vc-graphics/vc-graphics-hardfp.bb
meta-raspberrypi/recipes-graphics/vc-graphics/vc-graphics.bb
meta-raspberrypi/recipes-graphics/vc-graphics/vc-graphics.inc
meta-raspberrypi/recipes-graphics/wayland
meta-raspberrypi/recipes-graphics/wayland/weston_%.bbappend
meta-raspberrypi/recipes-graphics/xorg-xserver
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d/10-evdev.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d/98-pitft.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config/rpi/xorg.conf.d/99-calibration.conf
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xf86-config_0.1.bbappend
meta-raspberrypi/recipes-graphics/xorg-xserver/xserver-xorg_%.bbappend
meta-raspberrypi/recipes-kernel
meta-raspberrypi/recipes-kernel/linux-firmware
meta-raspberrypi/recipes-kernel/linux-firmware/files
meta-raspberrypi/recipes-kernel/linux-firmware/files/brcmfmac43430-sdio.bin
meta-raspberrypi/recipes-kernel/linux-firmware/files/brcfmac43430-sdio.txt
meta-raspberrypi/recipes-kernel/linux-firmware/linux-firmware_%.bbappend
meta-raspberrypi/recipes-kernel/linux
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi-dev.bb
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi.inc
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi_4.14.bb
meta-raspberrypi/recipes-kernel/linux/linux-raspberrypi_4.9.bb
meta-raspberrypi/recipes-multimedia
meta-raspberrypi/recipes-multimedia/gstreamer
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx/*.patch
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx_%.bbappend
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-plugins-bad_%.bbappend
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx-1.12
meta-raspberrypi/recipes-multimedia/gstreamer/gstreamer1.0-omx-1.12/*.patch
meta-raspberrypi/recipes-multimedia/omxplayer
meta-raspberrypi/recipes-multimedia/omxplayer/omxplayer
meta-raspberrypi/recipes-multimedia/omxplayer/omxplayer/*.patch
meta-raspberrypi/recipes-multimedia/omxplayer/omxplayer_git.bb
meta-raspberrypi/recipes-multimedia/x264
meta-raspberrypi/recipes-multimedia/x264/x264_git.bbappend
meta-raspberrypi/wic
meta-raspberrypi/wic/sdimage-raspberrypi.wks
```

1.3.1. Файлы лицензий

Файлы лицензий на уровне BSP имеют форму meta-bsp_root_name/bsp_license_file. Эти необязательные файлы указывают лицензионные требования для BSP. Тип файлов может зависеть от лицензии. Например, в Raspberry Pi BSP все лицензионные требования даны в файле COPYING.MIT. Лицензии могут быть MIT, BSD, GPLv* и пр. Требованиям к лицензиям даны в разделе Maintaining Open Source License Compliance During Your Product's Lifecycle [1].

1.3.2. Файл README

Файл meta-bsp_root_name/README содержит сведения о загрузке live-образов, которые могут включаться в каталог binary/, а также информацию, требуемую для сборки образа (по меньшей мере список зависимостей и данные сопровождающего BSP).

1.3.3. Файл README.sources

Файл meta-bsp_root_name/README.sources содержит информацию о местоположении исходных файлов BSP, использованных для сборки образов (при наличии), которые находятся в meta-bsp_root_name/binary. Эти образы распространяются с BSP. Информация в файле также помогает найти метаданные, использованные для генерации

образов, распространяемых с BSP. Если каталог binary отсутствует или не содержит образов, файл README.sources не имеет смысла и обычно не включается.

1.3.4. Готовые двоичные файлы

Необязательная область meta-bsp_root_name/binary/bootable_images может включать собранные образы ядер и пользовательских файловых систем, распространяемых с BSP и подходящих для целевой системы. Этот каталог обычно содержит live-образы и образы с поддержкой графики (например, Sato), когда создаётся архив BSP для распространения через сайт YP. Эти ядра и образы можно применять для ускорения работы.

Тип двоичных файлов существенно зависит от аппаратной платформы. В файле README уровня BSP следует описывать использование этих образов с целевым оборудованием. Кроме того, в файле README.sources следует указывать местоположение исходного кода, использованного для сборки образов, а также сведения о метаданных.

1.3.5. Файл конфигурации уровня

Файл meta-bsp_root_name/conf/layer.conf описывает файловую структуру уровня и содержит информацию о её использовании системой сборки. Обычно применяется стандартный шаблон файла, пример которого показан ниже с использованием bsp в качестве замены имени реального уровня BSP (bsp_root_name в шаблоне).

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

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

BBFILE_COLLECTIONS += "bsp"
BBFILE_PATTERN_bsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_bsp = "6"

LAYERDEPENDS_bsp = "intel"
```

Для иллюстрации подстановки строк ниже приведены операторы из файла conf/layer.conf для Raspberry Pi.

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

# We have a recipes directory containing .bb and .bbappend files, add to BBFILES
BBFILES += "${LAYERDIR}/recipes*/*/*.bb \
            ${LAYERDIR}/recipes*/*/*.bbappend"

BBFILE_COLLECTIONS += "raspberrypi"
BBFILE_PATTERN_raspberrypi := "^${LAYERDIR}/"
BBFILE_PRIORITY_raspberrypi = "9"

# Additional license directories.
LICENSE_PATH += "${LAYERDIR}/files/custom-licenses"
     ...
```

Этот файл просто указывает BitBake каталоги заданий и конфигурации. Файл нужен для распознавания BSP системой сборки OE.

1.3.6. Опции настройки оборудования

Файлы meta-bsp_root_name/conf/machine/*.conf содержат информацию о машине в понятном системе сборки формате. На каждом уровне BSP должен быть хотя бы один файл конфигурации машины. При поддержке нескольких машин может использоваться множество файлов. Имена файлов соответствуют значениям в переменной MACHINE.

Эти файлы определяют используемый пакет ядра (PREFERRED_PROVIDER в virtual/kernel), аппаратные драйверы для включения в разные типы образов, требуемые специальные программы и данные загрузчика, а также специальные требования к формату образов. Файлы могут также включать тонкую настройку оборудования, которая обычно применяется для определения архитектуры пакетов и специальных флагов оптимизации. Файлы тонкой настройки размещаются в каталоге meta/conf/machine/include дерева исходного кода. Например, многие файлы tune-* (tune-arm1136jf-s.inc, tun-1586-nlp.inc и т. п.) находятся в каталоге poky/meta/conf/machine/include.

Для использования включаемых файлов нужно просто указать их в файле конфигурации машины. Например, файл Raspberry Pi BSP raspberrypi3.conf содержит строку include conf/machine/include/rpi-base.inc.

1.3.7. Другие файлы BSP

Остальные файлы размещаются на уровне BSP в meta-bsp_root_name/recipes-bsp/*. Этот необязательный каталог содержит разные файлы заданий для BSP, среди которых наиболее важны файлы formfactor. Например, в Raspberry Pi BSP имеется файл formfactor_0.0.bbappend, который расширяет задание. Кроме того, имеются специфические для машины настройки, которые определяются файлом machconfig, например, для Raspberry Pi BSP machconfig имеет вид

```
HAVE_TOUCHSCREEN=0
HAVE_KEYBOARD=1

DISPLAY_CAN_ROTATE=0
DISPLAY_ORIENTATION=0
DISPLAY_DPI=133
```

Если BSP не включает записи formfactor, используются параметры из принятого по умолчанию файла formfactor, используемого основным заданием meta/recipes-bsp/formfactor/formfactor_0.0.bb в дереве исходного кода.

1.3.8. Файлы поддержки дисплеев

Файлы поддержки дисплея находятся в meta-bsp_root_name/recipes-graphics/*. Этот необязательный каталог содержит задания для BSP, имеющих особые требования по поддержке графики (все файлы BSP для поддержки дисплеев).

1.3.9. Конфигурация ядра Linux

Файлы конфигурации ядра на уровне BSP включают

```
meta-bsp_root_name/recipes-kernel/linux/linux*.bbappend
meta-bsp_root_name/recipes-kernel/linux/*.bb
```

Файлы дополнения (*.bbappend) меняют основное задание для ядра, используемое при сборке образа. Файлы *.bb содержат представленные разработчиком задания для ядра. Эта часть иерархии BSP может включать оба типа файлов, хотя на практике зачастую применяется лишь один.

Обычно для BSP применяется имеющееся в YP ядро из каталога исходных кодов meta/recipes-kernel/linux. Зависящие от машины изменения можно внести в файл добавления с похожим на задание именем, который размещается на уровне BSP для целевого устройства (например, в каталоге meta-bsp_root_name/recipes-kernel/linux). Предположим, что для сборки ядра служит задание linux-yocto_4.4.bb, т. е. оно указано в переменных PREFERRED_PROVIDER и PREFERRED_VERSION файла bsp_root_name.conf в форме

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "4.4%"
```

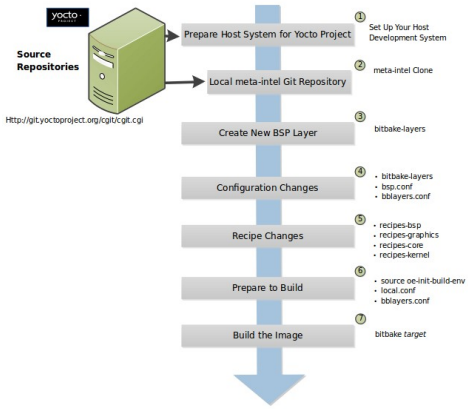
При использовании принятого по умолчанию предпочтительного провайдера переменная PREFERRED_PROVIDER не включается в файл bsp_root_name.conf. Файл linux-yocto_4.4.bbappend добавляет конкретные установки BSP в ядро, настраивая его для определённого устройства. Описание файлов дополнения приведено в разделе Creating the Append File [2].

Другим вариантом является организация своего задания для ядра в составе BSP. Примером может служить Raspberry Pi BSP, где в каталоге recipes-kernel/linux содержатся файлы трёх заданий и включаемый файл.

```
linux-raspberrypi-dev.bb
linux-raspberrypi.inc
linux-raspberrypi_4.14.bb
linux-raspberrypi_4.9.bb
```

1.4. Разработка BSP

Здесь кратко рассмотрена процедура создания BSP на примере уровня meta-intel (см. также раздел 1.8. Создание уровня BSP с помощью сценария bitbake-layers). Процесс создания BSP показан на рисунке и кратко описан ниже.



- *Организация хоста для работы с YP* в соответствии с разделом Preparing the Build Host [1].
- Организация локального репозитория meta-intel для получения доступа к уровням, которые можно использовать при создании своего BSP (см. раздел 1.2. Подготовка хоста сборки для работы с уровнем BSP).
- *Создание своего уровня BSP с использованием сценария bitbake-layers.* Уровни обеспечивают идеальное решение для изоляции и хранения работы с конкретным оборудованием. Уровень представляет собой просто хранилище (каталог) для заданий и файлов конфигурации BSP. Фактически BSP является просто одним из типов уровней. Проще всего создать совместимый с YP уровень BSP используя сценарий bitbake-layers (см. раздел 1.8. Создание уровня BSP с помощью сценария bitbake-layers).

  Другим примером использования уровней являются приложения. Предположим, что создаётся приложение, которое включает библиотеку или иную зависимость при компиляции и работе. В этом случае уровень будет содержать задания, где указаны все эти зависимости. Уровень по сути является изолированной областью, содержащей всю относящуюся к делу информацию, которую нужно знать системе сборки OE. Уровни более подробно описаны в разделе The Yocto Project Layer Model и Understanding and Creating Layers [1], а также в разделе 1.1. Уровни BSP.

  - В выпуске YP имеется 5 эталонных BSP на уровне poky/meta-yocto-bsp:
    - Texas Instruments Beaglebone (beaglebone-yocto);
    - Freescale MPC8315E-RDB (mpc8315e-rdb);
    - Ubiquiti Networks EdgeRouter Lite (edgerouter);
    - 2 платформы общего назначения IA (genericx86 и genericx86-64).
  - Три базовых Intel BSP имеются в выпуске YP на уровне meta-intel:
    - intel-core2-32, оптимизированный для семейства Core2, а также процессоров до Silvermont;

  - intel-corei7-64, оптимизированный для Nehalem и последующих процессоров Core и Xeon, а также Silvermont и поздних Atom, таких как Baytrail;
  - intel-quark, оптимизированный для плат Intel Galileo gen1 и gen2.

  При создании уровня BSP следует соблюдать стандартную схему, описанную в разделе 1.3. Пример структуры файлов, обращая внимание на структуру заданий и файлов конфигурации. В качестве примеров можно использовать поддерживаемые BSP уровня meta-intel в дереве исходного кода.

- *Настройка конфигурации уровня BSP.* В стандартной структуре BSP файлы, которые нужно редактировать, размещаются в каталоге conf и нескольких каталогах recipes-* уровня BSP. Изменения конфигурации включают размещение уровня в локальной системе и указывают используемое ядро. При запуске сценария bitbake-layers можно в интерактивном режиме задать многие настройки BSP (клавиатура, сенсорный экран и т. п.).
- *Изменение заданий нового уровня BSP,* включая редактирование файлов *.bb, удаление ненужных заданий и добавления новых заданий и/или файлов дополнения (.bbappend) для поддержки оборудования.
- *Подготовка к сборке.* После внесения изменений в уровень BSP остаётся организовать среду сборки с помощью сценария oe-init-build-env и убедиться в корректности содержимого файлов conf/local.conf и conf/bblayers.conf. Система сборки OE должна знать о новом уровне (см. раздел Enabling Your Layer [1]).
- *Сборка образа.* Система сборки OE использует BitBake для создания образов заданного типа (см. руководство пользователя BitBake). Процесс сборки поддерживает разные типов образов, описанные в разделе Images [3].

1.5. Требования и рекомендации для выпуска BSP

Существует ряд требований к выпускаемым BSP в части их совместимости с YP, описанных в этом разделе.

1.5.1. Требования к выпускаемым BSP

Перед рассмотрением требований к BSP следует учесть некоторые аспекты.

- Предполагается, что уровень BSP корректно сформирован и пригоден для добавления в YP. Рекомендации по созданию уровней приведены в разделе 1.1. Уровни BSP и Understanding and Creating Layers [1].
- Применение требований не зависит от способа упаковки BSP. Пример требований к упаковке и распространению приведён на странице Third Party BSP Release Process.
- Требования к BSP в том виде, как уровень представляется разработчику совершенно не зависят от формы выпускаемого BSP. Например, метаданные BSP могут содержаться в репозитории Git и иметь структуру каталогов, совершенно отличающуюся от официально выпущенного BSP.
- Не требуется включать какие-то определённые пакеты или модификации в BSP сверх требований общего соответствия YP. Например, не задаётся требований использовать определённую версию ядра в BSP.

Ниже приведены требования к BSP для выпуска в составе YP.

- *Имя уровня* должно соответствовать стандартам YP, как указано в разделе 1.1. Уровни BSP.
- *Структура файловой системы.* По возможности следует использовать на уровне BSP имена каталогов, указанные в файле recipes.txt из каталога poky/meta в дереве источников или уровня openembedded-core (http://git.openembedded.org/openembedded-core/tree/meta). Следует размещать файлы заданий *.bb и дополнения *.bbappend в каталогах recipes-* по функциональным областям, указанным в файле recipes.txt. Если в этом файле нет подходящей категории для того или иного задания, можно создать свой каталог recipes-*.

  Внутри категорий recipes-* структура должна соответствовать репозиторию Git openembedded-core или дереву исходных кодов poky. Связанные файлы следует размещать в подходящем каталоге recipes-* в соответствии с функциями задания. Сами задания следует оформлять в соответствии с руководством OpenEmbedded Style.
- *Файл лицензий* должен размещаться в каталоге meta-bsp_root_name. Лицензия охватывает метаданные BSP в целом и её нужно указать, поскольку принятой по умолчанию лицензии не задано. Примером может служить файл COPYING.MIT для Raspberry Pi BSP уровня meta-raspberrypi.
- *Файл README* должен размещаться в каталоге meta-bsp_root_name. Примером может служить файл README.md для Raspberry Pi BSP уровня meta-raspberrypi. Файл должен включать по меньшей мере перечисленное ниже.
  - Краткое описание оборудования для BSP.
  - Список всех зависимостей уровня BSP, который обычно включает другие уровни, требуемые для сборки BSP, однако может включать и другие зависимости.
  - Особые требования лицензирования, например, переменные, требуемые для EULA, или инструкции по сборке и распространению двоичных файлов, собранных с метаданными BSP.
  - Имя и контактные данные сопровождающего уровень BSP (человек, которому можно отправлять правки и вопросы), в соответствии с разделом Submitting a Change to the Yocto Project [1].
  - Инструкции по сборке BSP с использованием данного уровня.
  - Инструкции по загрузке образа, собранного из уровня BSP.
  - Инструкции по загрузке двоичных образов, если они имеются в каталоге.
  - Информация об известных ошибках при сборке или загрузке двоичных образов BSP.

- *Файл README.sources* должен включаться в каталог meta-bsp_root_name при наличии в BSP двоичных образов (каталог binary). Файл указывает местоположение исходного кода для сборки двоичных файлов.
- *Файл конфигурации уровня* conf/layer.conf в каталоге meta-bsp_root_name, указывающий уровень BSP для системы сборки.
- *Файлы конфигурации машины* в каталоге conf/machine/bsp_root_name.conf внутри meta-bsp_root_name. Эти файлы определяют целевые машины, для которых предназначен уровень BSP. Если BSP поддерживает несколько вариантов машины, нужно описать каждый вариант в файле README уровня BSP. Не следует включать файлы конфигурации для разнотипных машин, лучше создавать для них свои уровни BSP.

Разработчик может по-своему структурировать рабочие репозитории файлов BSP и готовить их к выпуску, пользуясь своими сценариями или иными средства. Этот вопрос выходит за рамки документа.

1.5.2. Рекомендации для выпуска BSP

- *Загружаемые образы.* Выпускаемые BSP могут включать загружаемые образы, что позволяет пользователям легко проверить BSP на своём оборудовании. В некоторых случаях включение загружаемых образов может оказаться неудобным и тогда можно сделать два варианта BSP, включая такие образы лишь в один из них. Это позволит избежать загрузки ненужных файлом теми пользователями, которым они не требуются.

  Если нужно распространять BSP с загружаемыми образами или собирать ядро и файловые системы, предназначенные для загрузки BSP с целью оценки, следует помещать образы и результаты сборки в каталог binary/ внутри meta-bsp_root_name. Если включенный в BSP двоичный образ собран с использованием программ, распространяемых по лицензии GPL или иной лицензии для открытого кода, нужно выполнить все требования лицензии в части распространения исходного кода программ.
- *Использование ядра Yocto Linux.* Задания для ядра в BSP следует основывать на ядрах Yocto Linux, что позволит снизить издержки на сопровождение BSP. Ядра доступны в Source Repositories.

1.6. Настройка задания для BSP

При планировании задания для конкретного BSP следует учитывать приведённые ниже требования.

- Для изменённого задания следует создать файл дополнения *.bbappend, как описано в разделе Using .bbappend Files in Your Layer [1].
- Структура каталогов уровня BSP, поддерживающего машину, должна быть устроена так, чтобы система сборки OE могла найти её (см. пример ниже).
- Файл дополнения следует поместить в каталог, соответствующий имени машины, в подходящем каталоге структуры уровня BSP (recipes-bsp, recipes-graphics, recipes-core и т. п.).
- Относящиеся к BSP файлы следует размещать в подходящих каталогах внутри уровня BSP. Например, если уровень поддерживает несколько типов машины, нужно обеспечить иерархию каталогов, разделяющую файлы по машинам. Если поддерживается лишь одна машина, такая иерархия не нужна и все файлы могут размещаться в каталоге для машины.

Ниже приведён пример настройки задания путём добавления файла init-ifupdown_1.0.bb для машины xyz в уровне BSP, поддерживающем несколько разных машин.

1. Следует включить в файл init-ifupdown_1.0.bbappend строку вида FILESEXTRAPATHS_prepend := "${THISDIR}/files:". Файл дополнения должен размещаться в каталоге meta-xyz/recipes-core/init-ifupdown.
2. Создаётся файл конфигурации интерфейсов на уровне BSP и помещается в meta-xyz/recipes-core/init-ifupdown/files/xyz-machine-one/interfaces. Если уровень meta-xyz не включает множества машин, можно поместить файл конфигурации интерфейсов в каталог meta-xyz/recipes-core/init-ifupdown/files/interfaces. Переменная FILESEXTRAPATHS в файле дополнения расширяет путь поиска системы сборки для нахождения файлов, поэтому каталог files нужно разместить там же, где находится файл дополнения.

1.7. Вопросы лицензирования BSP

В некоторых случаях BSP содержит отдельно лицензируемую интеллектуальную собственность (IP), поэтому требуется принять условия коммерческого или иного использования, требующие явного лицензионного соглашения с конечным пользователем (EULA). Когда такое соглашение принято, система сборки OE сможет использовать соответствующие компоненты при создании образа BSP. Если BSP доступен в собранном виде, можно загрузить образ после согласия с лицензией или EULA.

Могут возникать ситуации, когда важные для работы системы компоненты могут не иметь свободных (или бесплатных) замен, а без этих компонент система не будет работать. И напротив, могут возникать ситуации, когда другие лицензируемые компоненты пригодные для использования или необязательные имеют бесплатную замену, которую можно применять вместо согласия на отдельно лицензируемые компоненты. Даже для важных в системе компонент могут быть свободные замены, которые не обеспечивают полной функциональности, но вполне пригодны.

В случаях, когда возможна замена свободными компонентами с сохранением функциональности, выбор DOWNLOADS на вкладке SOFTWARE сайта YP делает доступными “усечённые” BSP, свободные от ограничений IP. В таких случаях замену можно применять напрямую без дополнительного лицензирования. При наличии “усечённых” BSP они именуются иначе, нежели полные варианты BSP, и обычно предполагают соответствие требованиям системы. Если таких вариантов нет или они недостаточно функциональны, придётся использовать “обременённую” версию.

В системе сборки OE имеется иной метод выполнения лицензионных требований с обременением, описанный ниже.

1. *Использование переменной LICENSE_FLAGS для заданий, имеющих пакеты с коммерческими и иными специальными лицензиями.* Для каждого из таких заданий можно указать соответствующую строку лицензии в файле local.conf (переменная LICENSE_FLAGS_WHITELIST), что будет указывать согласие с лицензией. В

результате система сборки может собирать соответствующее задание и включать компоненты в образ. Использование переменных описано в разделе Enabling Commercially Licensed Recipes [1]. Если заданий не указано в переменной LICENSE_FLAGS_WHITELIST, сборка будет остановлена с выводом списка заданий, которые включены в создание образа, но не указаны в LICENSE_FLAGS_WHITELIST. После указания таких заданий в переменной следует повторить сборку.

2. *Использование собранной версии BSP,* доступной по ссылке DOWNLOADS вкладки SOFTWARE на сайте YP. Можно загрузить архивы BSP с фирменными компонентами после согласия с лицензионными требованиями каждого из пакетов в процессе загрузки. Такой способ получения BSP обеспечивает доступ к образу сразу после принятия лицензионного соглашения, представленного на сайте. При самостоятельной сборке образов с использованием заданий из таких BSP нужно будет создать соответствующую LICENSE_FLAGS_WHITELIST, соответствующую заданиями из BSP.

   Предварительно собранные образы включают ограничения по времени использования в ядре (10 дней), после чего система перезагружается. Это предотвращает дальнейшее распространение образа и для снятия ограничения образ потребуется собрать заново.

1.8. Создание уровня BSP с помощью сценария bitbake-layers

Сценарий bitbake-layers автоматизирует создание уровня BSP, который отличается от других уровней наличием файла конфигурации машины. Кроме того, такие уровни обычно имеют задание для ядра или файл дополнения к имеющемуся заданию. Операции создания уровня BSP перечислены ниже.

- *Создание обычного уровня* с помощью команды bitbake-layers create-layer, как описано в разделе Creating a General Layer Using the bitbake-layers Script [1].
- *Создание файла конфигурации,* который нужен на каждом уровне и указывает местоположение заданий уровня, приоритет уровня и т. п. Примеры файлов layer.conf files представлены на странице YP Source Repositories. Для получения примеров можно выбрать уровень (скажем, meta-ti) и загрузить его конфигурационный файл.
- *Создание файла конфигурации машины* в форме conf/machine/bsp_root_name.conf. Примером могут служить файлы из каталога meta-yocto-bsp/conf/machine, а также каталоги производителей meta-ti и meta-freescale.
- *Подготовка задания для ядра* в форме recipes-kernel/linux с использованием файла добавления или созданием нового файла (например, yocto-linux_4.12.bb). Образцы можно найти в упомянутых выше уровнях, а рекомендации по подготовке заданий для ядра даны в разделе Modifying an Existing Recipe [2].

Далее уровень BSP рассматривается на примере эталонного YP BSP для Beaglebone с уровня meta-yocto-bsp.

1.8.1. Пример конфигурации уровня BSP

Каталог conf внутри уровня содержит файл layer.conf, пример которого представлен 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 += "yoctobsp"
BBFILE_PATTERN_yoctobsp = "^${LAYERDIR}/"
BBFILE_PRIORITY_yoctobsp = "5"
LAYERVERSION_yoctobsp = "4"
LAYERSERIES_COMPAT_yoctobsp = "zeus"
```

Дополнительные примеры конфигурационных файлов BSP можно найти в Source Repositories. Подробное описание процесса создания файла конфигурации приведено в п. 3 раздела Creating Your Own Layer [1].

1.8.2. Пример конфигурации машины BSP

Как отмечено выше, наличие файла конфигурации машины является отличительной чертой уровней BSP. Этот файл размещается в каталоге bsp_layer/conf/machine/. Например, файл конфигурации плат BeagleBone и BeagleBone Black хранится в каталоге poky/meta-yocto-bsp/conf/machine и называется beaglebone-yocto.conf:

```
#@TYPE: Machine
#@NAME: Beaglebone-yocto machine
#@DESCRIPTION: Reference machine configuration for http://beagleboard.org/bone and http://beagleboard.org/black boards

PREFERRED_PROVIDER_virtual/xserver ?= "xserver-xorg"
XSERVER ?= "xserver-xorg \
           xf86-video-modesetting \
           "

MACHINE_EXTRA_RRECOMMENDS = "kernel-modules kernel-devicetree"

EXTRA_IMAGEDEPENDS += "u-boot"

DEFAULTTUNE ?= "cortexa8hf-neon"
include conf/machine/include/tune-cortexa8.inc

IMAGE_FSTYPES += "tar.bz2 jffs2 wic wic.bmap"
EXTRA_IMAGECMD_jffs2 = "-lnp "
WKS_FILE ?= "beaglebone-yocto.wks"
IMAGE_INSTALL_append = " kernel-devicetree kernel-image-zimage"
do_image_wic[depends] += "mtools-native:do_populate_sysroot dosfstools-native:do_populate_sysroot"

SERIAL_CONSOLES ?= "115200;ttyS0 115200;ttyO0"
SERIAL_CONSOLES_CHECK = "${SERIAL_CONSOLES}"
```

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "5.0%"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "am335x-bone.dtb am335x-boneblack.dtb am335x-bonegreen.dtb"
KERNEL_EXTRA_ARGS += "LOADADDR=${UBOOT_ENTRYPOINT}"

SPL_BINARY = "MLO"
UBOOT_SUFFIX = "img"
UBOOT_MACHINE = "am335x_evm_defconfig"
UBOOT_ENTRYPOINT = "0x80008000"
UBOOT_LOADADDRESS = "0x80008000"

MACHINE_FEATURES = "usbgadget usbhost vfat alsa"

IMAGE_BOOT_FILES ?= "u-boot.${UBOOT_SUFFIX} MLO zImage am335x-bone.dtb am335x-boneblack.dtb am335x-bonegreen.dtb"
```

Переменные определяют машинозависимые свойства, например, специальные пакеты для машины, настройки, тип собираемого ядра, конфигурация U-Boot.

Ниже приведены некоторые пояснения из примеров файла конфигурации эталонной машины BeagleBone, а более полную информацию можно найти в разделе Yocto Project Variables Glossary [3].

- PREFERRED_PROVIDER_virtual/xserver - задание, обеспечивающее virtual/xserver при наличии нескольких провайдеров. В данном случае в качестве virtual/xserver служит задание xserver-xorg из poky/meta/recipes-graphics/xorg-xserver.
- XSERVER - пакеты, которые следует установить для поддержки X-сервера и драйверов для машины. В примере устанавливаются пакеты xserver-xorg и xf86-video-modesetting.
- MACHINE_EXTRA_RRECOMMENDS - список определяемых машиной пакетов, которые не важны для загрузки образа (т. е. Не будет возникать отказа при их отсутствии), однако нужных при работе. Имеется много переменных MACHINE, помогающих настроить конкретные компоненты оборудования.
- EXTRA_IMAGEDEPENDS - список заданий для сборки, которые не предоставляют пакетов, устанавливаемых в корневую файловую систему, но от которых зависит сборка образа. В данном случае для сборки требуется задание U-Boot.
- DEFAULTTUNE - настройки для оптимизации производительности машины, CPU и приложений. Эти свойства, называемые настройками производительности, размещаются на уровне OpenEmbedded-Core (OE-Core), например, в poky/meta/conf/machine/include. В примере используется настройка cortexa8hf-neon. Оператор include в файле conf/machine/include/tune-cortexa8.inc обеспечивает дополнительны возможности настройки.
- IMAGE_FSTYPES задаёт формат, применяемый системой сборки OE при создании корневой файловой системы. В примере поддерживаются 4 типа образов.
- EXTRA_IMAGECMD задаёт опции команды создания образа. В примере "-lnp " задаёт создание образа JFFS2.
- WKS_FILE указывает местоположение файла Wic kickstart, используемого системой сборки OE для создания образа с разделами (image.wic).
- IMAGE_INSTALL задаёт пакеты, устанавливаемые в образ через класс image.
- do_image_wic[depends] указывает задачи, создаваемые в процессе сборки. В примере задача зависит от инструментов, применяемых для создания sysroot в процессе сборки образа Wic.
- SERIAL_CONSOLES включает последовательные консоли (TTY) для getty. В примере задана скорость 115200 и имя устройства ttyO0.
- PREFERRED_PROVIDER_virtual/kernel указывает задание, обеспечивающее virtual/kernel при наличии нескольких провайдеров. В примере таким заданием служит linux-yocto из каталога recipes-kernel/linux.
- PREFERRED_VERSION_linux-yocto указывает версию задания, используемого для сборки ядра (5.0).
- KERNEL_IMAGETYPE - тип ядра для устройства. В примере система сборки OE создаёт образ типа zImage.
- KERNEL_DEVICETREE указывает имена создаваемых файлов дерева устройств ядра Linux (*.dtb). В примере включены все деревья разных устройств BeagleBone.
- KERNEL_EXTRA_ARGS указывает дополнительные аргументы, которые система сборки OE передаёт при компиляции ядра. В примере это "LOADADDR=${UBOOT_ENTRYPOINT}".
- SPL_BINARY определяет тип вторичного загрузчика программ (SPL[1]). В примере это MLO (Multimedia card Loader), поскольку плата BeagleBone требует SPL этого типа. Дополнительные сведения о применении переменных SPL приведены в файле u-boot.inc.
- UBOOT_* - определяет различные конфигурации U-Boot, требуемые для загрузки образа U-Boot. В примере образ U-Boot нужен для загрузки устройства BeagleBone. Задан также ряд дополнительных переменных:
  - UBOOT_SUFFIX указывает созданное расширение U-Boot;
  - UBOOT_MACHINE указывает значение, передаваемое команде make при сборке образа U-Boot;
  - UBOOT_ENTRYPOINT задаёт точку входа в образ U-Boot;
  - UBOOT_LOADADDRESS задаёт адрес загрузки образа U-Boot.
- MACHINE_FEATURES задаёт список поддерживаемых аппаратных свойств BeagleBone. В примере это "usbgadget usbhost vfat alsa".

[1]Secondary Program Loader.

- IMAGE_BOOT_FILES - указывает файлы, устанавливаемые в коренной раздел устройства при подготовке образа с использованием Wic и плагина bootimg-partition source.

1.8.3. Пример задания для ядра в BSP

Задание для ядра, применяемое при сборке ядра для устройства BeagleBone, указано в конфигурации машины.

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "5.0%"
```

Каталог meta-yocto-bsp/recipes-kernel/linux на уровне содержит метаданные, используемые для сборки ядра. В данном случае файл дополнения (linux-yocto_5.0.bbappend) переопределяет имеющееся задание (linux-yocto_5.0.bb) из http://git.yoctoproject.org/cgit/cgit.cgi/poky/tree/meta/recipes-kernel/linux. Содержимое файла приведено ниже.

```
KBRANCH_genericx86  = "v5.0/standard/base"
KBRANCH_genericx86-64  = "v5.0/standard/base"
KBRANCH_edgerouter = "v5.0/standard/edgerouter"
KBRANCH_beaglebone-yocto = "v5.0/standard/beaglebone"
KBRANCH_mpc8315e-rdb = "v5.0/standard/fsl-mpc8315e-rdb"

KMACHINE_genericx86 ?= "common-pc"
KMACHINE_genericx86-64 ?= "common-pc-64"
KMACHINE_beaglebone-yocto ?= "beaglebone"

SRCREV_machine_genericx86    ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_genericx86-64 ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_edgerouter ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_beaglebone-yocto ?= "3df4aae6074e94e794e27fe7f17451d9353cdf3d"
SRCREV_machine_mpc8315e-rdb ?= "8b62af7f252af10588276802c4c6d7c502e875be"

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

LINUX_VERSION_genericx86 = "5.0.3"
LINUX_VERSION_genericx86-64 = "5.0.3"
LINUX_VERSION_edgerouter = "5.0.3"
LINUX_VERSION_beaglebone-yocto = "5.0.3"
LINUX_VERSION_mpc8315e-rdb = "5.0.3"
```

Этот файл дополнения работает для всех машин уровня meta-yocto-bsp. Соответствующие операторы добавляются в строку beaglebone-yocto и система сборки OE применяет эти операторы для переопределений в задании для ядра:

- KBRANCH указывает ветвь ядра, которая была проверена, исправлена (patch) и настроена при сборке;
- KMACHINE задаёт имя машины, известное ядру, которое может отличаться от имени в системе сборки OE;
- SRCREV указывает выпуск исходного кода, использованный для сборки образа;
- COMPATIBLE_MACHINE задаёт регулярное выражение, преобразуемое в имя одной или нескольких машин, совместимых с образом;
- LINUX_VERSION указывает версию Linux из kernel.org, используемую системой OE для сборки образа ядра.

Литература

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

[2] Yocto Project Linux Kernel Development Manual, https://www.yoctoproject.org/docs/3.0/kernel-dev/kernel-dev.html.

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

[4] Yocto Project Overview and Concepts Manual, https://www.yoctoproject.org/docs/3.0/overview-manual/overview-manual.html.

Перевод на русский язык

Николай Малых

nmalykh@protokols.ru