

## Руководство пользователя Buildroot

# Содержание

<b>Я Начинаю</b>	<b>1</b>
1 О Buildroot	2
2 Системные требования	3
2.1 Обязательные пакеты . . . . .	3
2.2 Дополнительные пакеты . . . . .	4
3 Получение Buildroot	5
4 Быстрый старт Buildroot	6
5 Ресурсы сообщества	8
 II Руководство пользователя	 9
6 Конфигурация Buildroot	10
6.1 Набор инструментов кросскомпиляции . . . . .	10
6.1.1 Внутренний бэкэнд цепочки инструментов . . . . .	11
6.1.2 Внешний бэкэнд инструментальной цепочки . . . . .	11
6.1.3 Создание внешней цепочки инструментов с помощью Buildroot . . . . .	12
6.1.3.1 Внешняя оболочка цепочки инструментов . . . . .	13
6.2 Управление /dev . . . . .	13
6.3 Инициализация системы . . . . .	14
7 Конфигурация других компонентов	16
8 Общее использование Buildroot	17
8.1 делать подкаски . . . . .	17
8.2 Понимание того, когда необходима полная перестройка . . . . .	18
8.3 Понимание того, как пересобирать пакеты . . . . .	19
8.4 Оффлайн сборки . . . . .	20

8.5 Строительство из дерева . . . . .	20
8.6 Переменные с реды . . . . .	21
8.7 Эффективная работа с образами файловой системы . . . . .	21
8.8 Подробная информация о пакетах . . . . .	22
8.9 Графическое отображение зависимостей между пакетами . . . . .	22
8.10 График продолжительности сборки . . . . .	23
8.11 Графическое отображение вклада пакетов в размер файловой системы . . . . .	23
8.12 Параллельная сборка верхнего уровня . . . . .	24
8.13 Расширенное использование . . . . .	25
8.13.1 Использование сгенерированной цепочки инструментов вне Buildroot . . . . .	25
8.13.2 Использование gdb в Buildroot . . . . .	25
8.13.3 Использование ccache в Buildroot . . . . .	26
8.13.4 Расположение загружаемых пакетов . . . . .	27
8.13.5 Цели make, специфичные для пакета . . . . .	27
8.13.6 Использование Buildroot во время разработки . . . . .	28
9. Индивидуальная настройка под конкретный проект . . . . .	30
9.1 Рекомендуемые структуры каталогов . . . . .	30
9.1.1 Реализация многоуровневых настроек . . . . .	31
9.2 Создание настроек вне Buildroot . . . . .	32
9.2.1 Макет внешнего дерева br2 . . . . .	33
9.2.1.1 Файл external.desc . . . . .	33
9.2.1.2 Файлы Config.in и external.mk . . . . .	33
9.2.1.3 Каталог configs/. . . . .	34
9.2.1.4 Каталог provide/. . . . .	34
9.2.1.5 Контент с свободной формой . . . . .	34
9.2.1.6 Дополнительные расширения ядра Linux . . . . .	34
9.2.1.7 Пример макета . . . . .	35
9.3 Создание конфигурации Buildroot . . . . .	38
9.4 Создание конфигурации других компонентов . . . . .	38
9.5 Настройка с генерированной целевой файловой системы . . . . .	39
9.5.1 Настройка прав доступа к файлам и прав собственности, а также добавление узлов пользовательских устройств . . . . .	40
9.6 Добавление пользовательских учетных записей . . . . .	41
9.7 Настройка после создания изображений . . . . .	41
9.8 Добавление патчей и хэшей для конкретного проекта . . . . .	41
9.8.1 Предоставление дополнительных исправлений . . . . .	41
9.8.2 Предоставление дополнительных хэшей . . . . .	42
9.9 Добавление пакетов, специфичных для проекта . . . . .	42
9.10 Краткое руководство по сокращению настроек, специфичных для вашего проекта . . . . .	43

10 Темы интеграции	45
10.1 Установленные пакеты . . . . .	45
10.2 Системный . . . . .	45
10.2.1 Демон DBus . . . . .	45
10.3 Использование SELinux в Buildroot . . . . .	45
10.3.1 Включение поддержки SELinux . . . . .	46
10.3.2 Тонкая настройка политики SELinux . . . . .	46
11 Часто задаваемые вопросы и устранение неполадок	47
11.1 Загрузка зависает после запуска сети . . . . .	47
11.2 Почему на целевой платформе нет компилятора? . . . . .	47
11.3 Почему на целевом устройстве нет файлов разработки? . . . . .	48
11.4 Почему нет документации по цели? . . . . .	48
11.5 Почему некоторые пакеты не видны в меню конфигурации Buildroot? . . . . .	48
11.6 Почему бы не использовать целевой каталог в качестве chroot-каталога? . . . . .	48
11.7 Почему Buildroot не генерирует двоичные пакеты (.deb, .ipkg...)?	48
11.8 Как ускорить процесс сборки? . . . . .	49
11.9 Как Buildroot поддерживает Y2038? . . . . .	50
12 Известные проблемы	51
13 Юридическое уведомление или лицензирование	52
13.1 Соблюдение лицензий с открытым исходным кодом . . . . .	52
13.2 Соблюдение лицензии Buildroot . . . . .	53
13.2.1 Патчи для пакетов . . . . .	53
14 За пределами Buildroot	54
14.1 Загрузите сгенерированные образы . . . . .	54
14.1.1 Загрузка NFS . . . . .	54
14.1.2 Live CD . . . . .	54
14.2 Chroot-окружение . . . . .	54
III Руководство разработчика	55
15 Как работает Buildroot	56
16 Стиль кодирования	57
16.1 Файл config.in . . . . .	57
16.2 Файл .mk . . . . .	57
16.3 Файл genimage.cfg . . . . .	59
16.4 Документация . . . . .	60
16.5 Поддержка криптографии . . . . .	60

17 Добавление поддержки для определенной платы	61
18 Добавление новых пакетов в Buildroot	62
18.1 Каталог пакетов . . . . .	62
18.2 Файлы конфигурации . . . . .	62
18.2.1 Файл config.in . . . . .	62
18.2.2 Файл config.in.host . . . . .	63
18.2.3 Выбор зависит от или выбрать . . . . .	63
18.2.4 Зависимости от параметров цели и цепочки инструментов . . . . .	65
18.2.5 Зависимости от ядра Linux, с образного buildroot . . . . .	67
18.2.6 Зависимости от управленияudev /dev . . . . .	67
18.2.7 Зависимости от функций, предоставляемых виртуальными пакетами . . . . .	67
18.3 Файл .mk . . . . .	68
18.4 Файл .hash . . . . .	68
18.5 Стартовый скрипт SNNfoo . . . . .	70
18.5.1 Запуск настройки скрипта . . . . .	71
18.5.2 Обработка PID-файла . . . . .	71
18.5.3 Остановка службы . . . . .	72
18.5.4 Перезагрузка конфигурации сервиса . . . . .	72
18.5.5 Коды возврата . . . . .	72
18.5.6 Ведение журнала . . . . .	72
18.6 Инфраструктура для пакетов со специальными системами сборки . . . . .	72
18.6.1 Учебник по универсальному пакету . . . . .	72
18.6.2 Ссылка на общий пакет . . . . .	74
18.7 Инфраструктура для пакетов на основе autotools . . . . .	80
18.7.1 Учебное пособие по пакету autotools . . . . .	80
18.7.2 Справочник по пакетам autotools . . . . .	80
18.8 Инфраструктура для пакетов на основе CMake . . . . .	81
18.8.1 Учебник по пакету cmake . . . . .	81
18.8.2 Ссылка на пакет cmake . . . . .	82
18.9 Инфраструктура для пакетов Python . . . . .	83
18.9.1 Учебник по пакету python . . . . .	83
18.9.2 Справочник по пакетам python . . . . .	84
18.9.3 Создание пакета Python из репозитория PyPI . . . . .	85
18.9.4 Серверная часть CFFI пакета python . . . . .	86
18.10 Инфраструктура для пакетов на основе LuaRocks . . . . .	86
18.10.1 luarocks-пакет учебник . . . . .	86
18.10.2 luarocks-ссылка на пакет . . . . .	87
18.11 Инфраструктура для пакетов Perl/CPAN . . . . .	87

18.11.1 Учебник по пакету perl	87
18.11.2 Справочник по perl-пакету	88
18.12 Инфраструктура для виртуальных пакетов	89
18.12.1 Учебник по виртуальным пакетам	89
18.12.2 Файл Config.in виртуального пакета	89
18.12.3 Файл виртуального пакета.mk	89
18.12.4 Файл Config.in поставщика	90
18.12.5 Файл .mk поставщика	90
18.12.6 Заметки о зависимости от виртуального пакета....	90
18.12.7 Примечания о зависимости от конкретного поставщика....	91
18.13 Инфраструктура для пакетов, использующих kconfig для файлов конфигурации	91
18.14 Инфраструктура для пакетов на основе арматуры	92
18.14.1 Учебное пособие по арматурному пакету	92
18.14.2 ссылка на пакет арматуры	93
18.15 Инфраструктура для пакетов на базе Waf.	93
18.15.1 Учебник по waf-пакету	93
18.15.2 ссылка на waf-пакет	94
18.16 Инфраструктура для пакетов на основе Meson.	95
18.16.1 Учебник по мезон-пакету	95
18.16.2 ссылка на мезонный пакет	95
18.17 Инфраструктура для рузовых пакетов.	96
18.17.1 Учебник по грузовым пакетам	96
18.17.2 ссылка на грузовое место	97
18.18 Инфраструктура для пакетов Go .	97
18.18.1 golang-package учебник	97
18.18.2 Справочник по golang-пакету	98
18.19 Инфраструктура для пакетов на основе QMake .	99
18.19.1 Учебник по qmake-package	99
18.19.2 Ссылка на qmake-пакет	99
18.20 Инфраструктура для пакетов, создающих модули ядра	100
18.20.1 Учебник по модулю ядра	100
18.20.2 ссылка на модуль ядра	101
18.21 Инфраструктура для документов asciidoc	102
18.21.1 asciidoc-документ учебник	102
18.21.2 asciidoc-ссылка на документ	103
18.22 Инфраструктура, специчная для пакета ядра Linux .....	104
18.22.1 linux-kernel-инструменты	104
18.22.2 Linux-ядро-расширения	105
18.23 Крючки, доступные на различных этапах сборки .	106

18.23.1 Использование <code>x</code> ука <code>POST_RSYNC</code>	107
18.23.2 Ук <code>Target-finalize</code>	107
18.24 Интеграция Gettext и взаимодействие с пакетами . . . . .	107
18.25 Советы и рекомендации . . . . .	108
18.25.1 Имя пакета, имя записи конфигурации и связь переменных <code>makefile</code> . . . . .	108
18.25.2 Как проверить стиль кодирования . . . . .	108
18.25.3 Как протестировать ваш пакет . . . . .	109
18.25.4 Как добавить пакет из GitHub . . . . .	110
18.25.5 Как добавить пакет из Gitlab . . . . .	111
18.26 Заключение . . . . .	111
<b>19. Патч пакета</b>	<b>112</b>
19.1 Представление исправлений . . . . .	112
19.1.1 Загружено . . . . .	112
19.1.2 Внутри Buildroot . . . . .	112
19.1.3 Глобальный каталог исправлений . . . . .	113
19.2 Как применяются яисправления . . . . .	113
19.3 Формат и лицензирование пакетов исправлений . . . . .	113
19.4 Дополнительная документация по исправлению . . . . .	114
<b>20 Загрузка инфраструктуры</b>	<b>115</b>
<b>21 Отладка Buildroot</b>	<b>116</b>
<b>22. Вклад в Buildroot</b>	<b>117</b>
22.1 Воспроизведение, анализ и исправление ошибок . . . . .	117
22.2 Анализ и исправление сбоев автосборки . . . . .	117
22.3 Проверка и тестирование исправлений . . . . .	118
22.3.1 Применение лоскутных заплаток из Patchwork . . . . .	119
22.4 Работа над пунктами из списка TODO . . . . .	119
22.5 Отправка исправлений . . . . .	119
22.5.1 Форматирование патча . . . . .	119
22.5.2 Подготовка серии патчей . . . . .	120
22.5.3 Сопроводительное письмо . . . . .	121
22.5.4 Патчи для веток обслуживания . . . . .	122
22.5.5 Журнал изменений исправлений . . . . .	122
22.6 Сообщение о проблемах / ошибках или получение помощи . . . . .	123
22.7 Использование фреймворка тестов времени выполнения . . . . .	124
22.7.1 Создание тестового случая . . . . .	125
22.7.2 Отладка тестового случая . . . . .	125
22.7.3 Тесты времени выполнения Gitlab CI . . . . .	126

23 Файл DEVELOPERS и get-developers	127
24 Выпуск к Инженерных	128
24.1 Выпуски . . . . .	128
24.2 Развитие . . . . .	128
<b>Приложение IV</b>	<b>129</b>
25 Документация по синтаксису Makedev	130
26 Документация по синтаксису Makeusers	132
26.1 Предосторожение относительно автоматических UID и GID . . . . .	133
27 Миграции от старых версий Buildroot	134
27.1 Общий подход . . . . .	134
27.2 Переход на 2016.11 . . . . .	134
27.3 Переход на 2017.08 . . . . .	135
27.4 Переход на 2023.11 . . . . .	135
27.5 Переход на 2024.05 . . . . .	136

Руководство Buildroot 2024.11.1 с создано 09.01.2025 14:44:45 UTC из ревизии git 31462e4169

Руководство по Buildroot написано разработчиками Buildroot. Оно лицензировано в соответствии с GNU General Public License, версия 2. Обратитесь к разделу [КОПИРОВАНИЕ](#) файла исходных кодов Buildroot для получения полного текста этой лицензии.

Авторские права © Работники Buildroot <[buildroot@buildroot.org](mailto:buildroot@buildroot.org)>

логотип.png

Часть 1

# Начиная

# Глава 1

## О Buildroot

Buildroot — это инструмент, который упрощает и автоматизирует процесс сборки полноценной системы Linux для встраиваемых систем, используя кросскомпиляцию.

Для достижения этой цели Buildroot способен генерировать цепочку инструментов кросскомпиляции, корневую файловую систему, образ ядра Linux и загрузчик для вашей цели. Buildroot можно использовать для любых комбинаций этих опций, независимо (например, можно использовать существующую цепочку инструментов кросскомпиляции и соберите только корневую файловую систему с помощью Buildroot).

Buildroot полезен в основном для любых, работающих с встроеннымами системами. Встроенные системы часто используют процессы, которые не обычные процессы x86, которые все привыкли иметь в своем ПК. Это могут быть процессы PowerPC, процессы MIPS, ARM процессы и т.д.

Buildroot поддерживает множество процессоров и их модификаций; также он поставляется с конфигурациями по умолчанию для нескольких доступных плат.<sup>1</sup> <sup>2</sup> или SDK поверх Buildroot.

---

<sup>1</sup>BSP: Пакет поддержки управления

<sup>2</sup>SDK: Комплект для разработки программного обеспечения

# Глава 2

## Системные требования

Buildroot предназначен для работы в системах Linux.

Хотя Buildroot сам построит большинство пакетов他自己, необязательных для компиляции, некоторые стандартные утилиты Linux, как ожидается уже установлены на системе. Ниже вы найдете обзор обязательных и дополнительных пакетов ( обратите внимание, что названия пакетов могут различаться в зависимости от дистрибутивов).

### 2.1 Обязательные пакеты

- Инструменты сборки:

- который
- но
- make (версия 3.81 или более поздняя)
- binutils
- build-essential (только для систем на базе Debian)
- диффутилы
- gcc (версия 4.8 или более поздняя)
- g++ (версия 4.8 или более поздняя)
- Баш
- глас тырь
- gzip
- bzip2
- perl (версия 5.8.7 или более поздняя)
- берет
- cpio
- распаковать
- rsync
- файл (должен находиться в /usr/bin/file)

- донашай трэй

- найти утилиты

- Инструменты извлечения исходного кода

- wget

## 2.2 Дополнительные пакеты

- Рекомендуемые зависимости:

Некоторые функции или утилиты в Buildroot, такие как legal-info или инструменты генерации графиков, имеют дополнительные зависимости. Хотя они не являются обязательными для просмотра сборки, они все равно настоятельно рекомендуются.

- python (версия 2.7 или более поздняя)

- Зависимости пользовательского интерфейса конфигуратора:

Для этих библиотек необходимо установить как перед выполнения, так и данные разработки, которые во многих дистрибутивах упакованы отдельно. Пакеты разработки обычно имеют суффиксы -dev или -devel.

- ncurses5 для использования интерфейса menuconfig – qt5

для использования интерфейса xconfig –

glib2, gtk2 и glade2 для использования интерфейса gconfig

- Инструменты извлечения исходного кода

В официальном дереве большинство пакетов извлекаются с помощью wget из ftp, http или https-мест. Несколько пакетов доступны только через систему контроля версий. Более того, Buildroot способен загружать исходные коды с помощью других инструментов, таких как git или scp (подробнее см. в Главе 20).

Если вы включаете пакеты с помощью любого из этих методов, вам нужно будет установить соответствующий инструмент на хост-системе:

- базар
  - завиток
  - резюме
  - мерзвец
  - рутный
  - SCP-объект
  - SFTP-сервер
  - подрывная деятельность

- Пакеты, связанные с Java, если необходимо построить Java Classpath для языка Java:

- Компилятор javac
  - Инструмент «банка»

- Инструменты создания документов:

- asciidoc, версия 8.6.3 или выше
  - w3m
  - python с модулем argparse (автоматически присутствует в 2.7+ и 3.2+)
  - dblatex (требуется только для руководства в формате PDF)

- Инструменты для создания графиков:

- graphviz для использования graph-depends и <pkg>-graph-depends – python-matplotlib для использования graph-build

- Инструменты статистики пакетов (pkg-stats):

- python-aiohttp

# Глава 3

## Получение Buildroot

Релизы Buildroot производятся каждые 3 месяца, в феврале, мае, августе и ноябре. Номера релизов имеют формат ГГГГ.ММ, например 2013.02, 2014.08.

Архивы релизов доступны по адресу <http://buildroot.org/downloads/>.

Для вашего удобства [Vagrantfile](#) доступен в файле support/misc/Vagrantfile в исходном дереве Buildroot, чтобы настроить виртуальную машину с необходимыми зависимостями для начала работы.

Если вы хотите настроить изолированную среду buildroot в Linux или Mac OS X, вставьте эту строку в свой терминал:

```
curl -O https://buildroot.org/downloads/Vagrantfile; vagrant up
```

Если вы используете Windows, вставьте это в PowerShell:

```
(новый-объект System.Net.WebClient).DownloadFile("https://buildroot.org/
downloads/Vagrantfile", "Vagrantfile");
бродяг а вверх
```

Если вы хотите следить за разработкой, вы можете использовать ежедневные снимки или сделать клон репозитория Git. Обратитесь к странице загрузки на сайте Buildroot для получения более подробной информации.

# Глава 4

## Быстрый старт Buildroot

Важно: вы можете и должны строить все как обычный пользователь. Нет необходимости быть root для начала работы с Buildroot.

Выполняя команды от имени обычного пользователя, вы защищаете свою систему от пакетов, ведущих себя некорректно во время компиляции и установки.

Первый шаг при использовании Buildroot — создание конфигурации. Buildroot имеет хороший инструмент конфигурации, о котором можно найти в [ядре Linux](#) или в [BusyBox](#).

Из каталога `abuildroot` запустите:

```
$ c делать menuconfig
```

для оригинального конфигуратора на основе curses, или

```
$ c делать nconfig
```

для нового конфигуратора на основе curses, или

```
$ c делать xconfig
```

для конфигуратора на основе Qt или

```
$ c делать gconfig
```

для конфигуратора на базе GTK.

Все эти команды "make" требуют сборки утилиты конфигурации (включая интерфейсы), поэтому вам может потребоваться установка пакетов "development" для соответствующих библиотек, используемых утилитами конфигурации. Обратитесь к Главе 2 для получения более подробной информации, в частности, [дополнительных требований](#) для получения зависимостей вашего бинарного интерфейса.

Для каждого пункта меню в инструменте конфигурации вы можете найти соответствующую справку, описывающую цель пункта. Обратитесь к Главе 6 для получения подробной информации о некоторых конкретных аспектах конфигурации.

После того, как все настроено, инструмент конфигурации генерирует файл `.config`, содержащий всю конфигурацию. Этот файл будет прочитан верхним уровнем `Makefile`.

Чтобы начать процесс с сборки, просто запустите:

```
$ c делать
```

По умолчанию Buildroot не поддерживает параллельную сборку верхнего уровня поэтому запуск `make -jN` не нужен. Однако существует экспериментальная поддержка параллельной сборки верхнего уровня, см. раздел [8.12](#).

Команда `make` обычно выполняет следующие шаги:

- загрузить исходные файлы (если это необходимо);

- настроить, сбрать и установить набор инструментов кросскомпиляции или просто импортировать внешний набор инструментов;
- настроить, сбрать и установить выбранные целевые пакеты;
- создать образ ядра, если выбрано;
- создать образ загрузчика, если выбрано;
- создать корневую файловую систему в выбранных форматах.

Вывод Buildroot хранится в одном каталоге, output/. Этот каталог содержит несколько подкаталогов:

- images/ где хранятся все образы (образ ядра, загрузчика и корневой файловой системы). Это те файлы, которые вам нужны для установки на целевую систему.
- build/, где собираются якобы компоненты (включая инструменты, необязательные Buildroot на хосте, и пакеты, скомпилированные для цели). Этот каталог содержит один подкаталог для каждого из этих компонентов.
- host/ содержит как инструменты, собранные для хоста, так и sysroot целевой epoch и инструментов. Первая предоставляет с собой установку инструментов, скомпилированных для хоста, которые необязательны для выполнения Buildroot, включая epoch инструментов кросскомпиляции. Последняя предоставляет с собой иерархию, похожую на иерархию корневой файловой системы. Она содержит заголовки и библиотеки всех пакетов пользовательского пространства, которые предоставляют и установливают библиотеки, используемые другими пакетами. Однако этот каталог не предназначен для использования в качестве корневой файловой системы для цели: он содержит множество файлов разработки, не разрезанных двоичных файлов и библиотек, которые делают его слишком большим для использования встроенной системы. Эти файлы разработки используются для компиляции библиотек и приложений для целевой платформы, которые зависят от других их библиотек.
- staging/ — это символический ссылка на целевую epoch инструментов sysroot внутри host/, которая существует для обратной совместимости.
- target/, который содержит почти полную корневую файловую систему для цели: все необязательные присутствуют, за исключением файлов устройств в /dev/ (Buildroot не может их создавать, поскольку Buildroot не запускается как root и не хочет запускаться как root). Кроме того, у него нет правильных разрешений (например, setuid для двоичного файла busybox). Поэтому этот каталог не следует использовать навешиваться на цель. Вместо этого следует использовать один из образов, созданный в каталоге images/. Если вам нужен извлеченный образ корневой файловой системы для загрузки по NFS, то используйте образ tarball, созданный в images/, и извлеките его как root. Сравнению с staging/, target/ содержит только файлы и библиотеки, необязательные для запуска выбранных целевых приложений: файлы разработки (заголовки и т. д.) отсутствуют, двоичные файлы удалены.

Эти команды, make menuconfig | nconfig | gconfig | xconfig и make, являются новыми, которые позволяют легко генерировать образы, соответствующие вашим потребностям, со всеми включенными вами функциями и приложениями.

Более подробная информация об использовании команды «make» приведена в разделе 8.1.

# Глава 5

## Ресурсы с сообщества

Как и любой проект с открытым исходным кодом, Buildroot имеет различные способы обмена информацией в соответствии с общим сообществом из-за ограничениями.

Каждый из этих способов может вас заинтересовать, если вы ищете помощь, обратитесь к Buildroot или внести свой вклад в проект.

### Mailing List

Buildroot имеет список рассылок для обсуждения и разработки. Это основной метод взаимодействия для пользователей и разработчиков Buildroot.

Только подписчики на рассылку Buildroot могут получать сообщения в этом списке. Вы можете подписаться через [информацию о рассылке страница](#).

Письма, отправленные в список рассылки, также доступны в архивах списков рассылок, доступных через [Mailman](#), или на [lore.kernel.org](#).

### IRC

IRC-канал Buildroot [#buildroot](#) размещен на [OFTC](#). Это полезное место, где можно быстро задать вопросы или обсудить определенные темы.

При запросе помощи в IRC поделитесь соответствующими журналами или фрагментами кода, используя сайт обмена кодом, например <https://paste.ack.tf/>.

Обратите внимание, что для некоторых вопросов размещение в списке рассылки может быть лучшим решением, поскольку это позволит их включить больше людей, как разработчиков, так и пользователей.

### Об ошибках

Buildroot можно сообщать через список рассылок или через [bug трекер Buildroot](#). Перед созданием отчета об ошибке ознакомьтесь с разделом [22.6](#).

### Wiki

Страница [Buildroot wiki](#) размещена на [eLinux wiki](#). Содержит несколько полезных ссылок, обзор прошедших и предстоящих событий, а также список TODO.

### Пэчворк

Patchwork — это веб-сайт леживания и правлений, разработанный для облегчения включения вкладов в проект с открытым исходным кодом.

Исправления, отправленные в список рассылок, «перехватываются» системой и появляются на веб-странице.

Любые комментарии, которые сообщаются на патч, также добавляются на страницу патча. Для получения дополнительной информации о Patchwork см. <http://jk.ozlabs.org/projects/patchwork/>.

Веб-сайт Buildroot's Patchwork основном предназначен для использования сопровождающим Buildroot, чтобы гарантировать, что исправления не будут пропущены. Он также используется ядренными исправлениями Buildroot (см. также раздел [22.3.1](#)). Однако, поскольку веб-сайт предоставляет исправления в соответствии с общим сообществом, он может быть полезен для других разработчиков Buildroot.

Интерфейс управления исправлениями Buildroot доступен по адресу <https://patchwork.ozlabs.org/project/buildroot/list/>.

## Час тъ 2

Руководство пользователя

# Глава 6

## Конфигурация Buildroot

Все параметры конфигурации в `make *config` имеют с правочный текст, содержащий подробную информацию о параметре.

Команды `make *config` также предлагают инструмент поиска. Прочитайте с правочное сообщение в различных меню интерфейса, чтобы узнать, как им пользоваться:

- в менюconfig инструмент поиска вызывается нажатием /;
- в xconfig инструмент поиска вызывается нажатием Ctrl + f.

Результат поиска показывает с правочное сообщение с соответствующими элементами. В менюconfig числа в левом столбце обес печивают ярлык для соответствующей записи. Просмотрите этот номер, чтобы напрямую перейти к записи или к содержащему меню, если запись не может быть выбрана из-за отсутствия щелчков зависимости.

Хотя структура меню и текст с правки к записям должны быть достаточно понятными, рядом требуют дополнительных пояснений, которые невозможнолегко вставить в тексте с правки, и поэтому расматриваются в следующих разделах.

### 6.1 Набор инструментов кросс-компиляции

Набор инструментов компиляции — это набор инструментов, который позволяет вам компилировать код для вашей системы. Он состоит из компилятора (в нашем случае gcc), бинарных утилит, таких как ассемблер и компоновщик (в нашем случае binutils) и стандартной библиотеки C (например, GNU Libc, uClibc-ng).

Система установлена на вашем компьютере, наверняка уже имеет набор инструментов компиляции, который вы можете использовать для компиляции приложений, работающих на вашем компьютере. Если вы используете ПК, ваш набор инструментов компиляции работает на процессоре x86 и генерирует код для процессора x86. В большинстве систем Linux набор инструментов компиляции использует GNU libc (glibc) в качестве стандартной библиотеки C. Этот набор инструментов компиляции называется «`x86-кросс-компиляционным набором инструментов`». Машина, на которой он работает и на которой вы работаете, называется «`x86-системой`»<sup>1</sup>.

Цепочка инструментов компиляции предоставляется вашим дистрибутивом, и Buildroot не имеет к ней никакого отношения (кроме использования ее для построения цепочки инструментов кросс-компиляции и других инструментов, которые запускаются на языках программирования).

Как было сказано выше, набор инструментов компиляции, который предоставляется вашей системой, работает на процессоре вашей хост-системы и генерирует код для него. Поскольку ваша система имеет другой процессор, вам нужен набор инструментов кросс-компиляции — набор инструментов компиляции, который работает на вашем хост-системе, но генерирует код для вашей целевой системы (и целевого процессора). Например, если ваша хост-система использует x86, а ваша целевая система использует ARM, обычный набор инструментов компиляции на вашем хосте работает на x86 и генерирует код для x86, в то время как набор инструментов кросс-компиляции работает на x86 и генерирует код для ARM.

Buildroot предоставляет два решения для цепочки инструментов кросс-компиляции:

- Внутренний бэкэнд цепочки инструментов, называемый Buildroot toolchain в интерфейсе конфигурации.

<sup>1</sup>Этот терминология отличается от той, что используется в GNU configure, где host — это машина, на которой будет работать приложение (обычно это тоже самое, что и цель).

- Внешний бэкэнд цепочки инструментов, называемый в интерфейсе конфигурации «Внешняя цепочка инструментов».

Выбор между этими двумя решениями осуществляется с помощью опции Toolchain Type в меню Toolchain. После выбора одного решения появляется ряд параметров конфигурации, которые подробно описаны в следующих разделах.

### 6.1.1 Внутренний бэкэнд цепочки инструментов

Внутренний бэкэнд цепочки инструментов — это бэкэнд, где Buildroot самостоятельно создает цепочку инструментов кросскомпиляции перед сборкой приложений и библиотек пользовательского пространства для целевой системы.

Этот бэкэнд поддерживает несколько библиотек C: [uClibc-ng](#), [glibk](#) и [musl](#).

После выбора этого бэкенда появляется ряд опций. Наиболее важные из них позволяют:

- Измените версию заголовков ядра Linux, используемых для построения цепочки инструментов. Этот пункт используется несолько раз для каждого ядра. В процессе построения цепочки инструментов кросскомпиляции соприкасается с библиотеками C. Эта библиотека предоставляет интерфейс между приложениями пользовательского пространства и ядром Linux. Чтобы знать, как «общаться» с ядром Linux, библиотеке С необходимо иметь доступ к заголовкам ядра Linux (т. е. файлам.h из ядра), которые определяют интерфейс между пользовательским пространством и ядром (системные вызовы, структуры данных и т. д.). Поскольку этот интерфейс обратно совместим, версия заголовков ядра Linux, используемых для построения цепочки инструментов, не обязательно должна точно соответствовать версии ядра Linux, которое вы собираетесь запустить на своей встраиваемой системе. Им нужно только иметь версию, равную или более старую, чем версия ядра Linux, которую вы собираетесь запустить. Если вы используете заголовки ядра, которые более поздние, чем ядро Linux, которое вы запускаете на своей встраиваемой системе, то библиотека C может использовать интерфейсы, которые не предстают ясным ядром Linux.
- Измените версию компилятора GCC, binutils и библиотеки C.
- Выберите несколько параметров цепочки инструментов (только uClibc): должна ли цепочка инструментов иметь поддержку RPC (использование NFS), поддержку широких символов, поддержку локали (для интернационализации), поддержку C++ или поддержку потоков. В зависимости от выбранных вами параметров изменится количество приложений и библиотек пользовательского пространства, отображаемых в меню Buildroot: многие приложения и библиотеки требуют включения определенных параметров цепочки инструментов. Большинство пакетов показывают комментарий, когда для включения этих пакетов требуется определенный параметр цепочки инструментов. При необходимости вы можете дополнительно уточнить конфигурацию uClibc, выполнив команду make uclibc-menuconfig. Однако обратите внимание, что все пакеты в Buildroot тестируются на соответствие конфигурации uClibc по умолчанию, включенной в Buildroot: если вы отклонитесь от этой конфигурации, удалив функции из uClibc, некоторые пакеты могут перестать собираться.

Стоит отметить, что всякий раз, когда один из этих вариантов изменяется, то вся цепочка инструментов и система должны быть перестроены. См. раздел 8.2.

Преимущества этого бэкенда:

- Хорошо интегрирован с Buildroot
- Быстро, с тратой только того, что необязательно

Недостатки этого бэкенда:

- Перестройка цепочки инструментов необязательна при выполнении make clean, что занимает время. Если вы пытаетесь сократить время сборки, расмотрите возможность использования внешнего бэкенда цепочки инструментов.

### 6.1.2 Внешний бэкэнд инструментальной цепочки

Внешний бэкэнд toolchain позволяет использовать существующие готовые кросскомпиляционные toolchains. Buildroot знает о ряде известных кросскомпиляционных toolchains (от Linaro для ARM, [Sourcey CodeBench](#) для ARM, x86-64, PowerPC и MIPS) и может загружать их автоматически, или ему можно указать пользовательский набор инструментов, доступный для загрузки или установленный локально.

Тогда у вас есть три решения по использованию внешней цепочки инструментов:

- Используйте предопределенный внешний профиль цепочки инструментов и позвольте Buildroot загружать, извлекать и устанавливать цепочку инструментов. Buildroot уже знает о нескольких цепочках инструментов CodeSourcery и Linaro. Просмотрите профиль цепочки инструментов в Toolchain из доступных. Это определено самое простое решение.
- Используйте предопределенный внешний профиль цепочки инструментов, но вместе с тем, чтобы Buildroot загружал и извлекал цепочку инструментов, вы можете указать Buildroot, где ваша цепочка инструментов уже установлена в вашей системе. Просмотрите профиль цепочки инструментов в Toolchain из доступных, снимите флагок Download toolchain automatic и заполните текстовую запись Toolchain path путем к вашей цепочке инструментов кросс-компиляции.
- Используйте полностью настраиваемую внешнюю цепочку инструментов. Это особенно полезно для цепочек инструментов, созданных с помощью crosstool-NG или с помощью самого Buildroot. Для этого выберите решение Custom toolchain в списке Toolchain. Вам необязательно заполнять параметры Toolchain path, Toolchain prefix и External toolchain C library. Затем вам нужно указать Buildroot, что поддерживает ваша внешняя цепочка инструментов. Если ваша внешняя цепочка инструментов использует библиотеку glibc, вам нужно только указать, поддерживает ли ваша цепочка инструментов C++ или нет, и имеет ли она встроенную поддержку RPC. Если ваша внешняя цепочка инструментов использует библиотеку uClibc, вам нужно указать Buildroot, поддерживает ли она RPC, wide-char, locale, вызов программ, потоки и C++. В начале выполнения Buildroot сообщит вам, соответствуют ли выбранные параметры конфигурации цепочки инструментов.

Наша поддержка внешних наборов инструментов была протестирована с наборами инструментов от CodeSourcery и Linaro, наборами инструментов, созданными [crosstool-NG](#), и toolchains, сгенерированные самим Buildroot. В общем, все toolchains, которые поддерживают функцию sysroot, должны работать. Если нет, не стесняйтесь обращаться разработчикам.

Мы не поддерживаем цепочки инструментов или SDK, с созданные OpenEmbedded или Yocto, поскольку эти цепочки инструментов не являются чистыми цепочками инструментов (т. е. только компилятор, binutils, библиотеки C и C++). Вместо этого эти цепочки инструментов состоят из очень большим набором предварительно скомпилированных библиотек и программ. Поэтому Buildroot не может импортировать sysroot цепочки инструментов, так как он будет сдерживать сдвиг магнита предварительно скомпилированных библиотек, которые обычно создаются Buildroot.

Мы также не поддерживаем использование дистрибутивной инструментальной цепочки (т. е. библиотеки gcc/binutils/C, установленной вашим дистрибутивом) в качестве инструментальной цепочки для сборки и программы для ядра. Это связано с тем, что ваша дистрибутивная инструментальная цепочка не является «чистой» инструментальной цепочкой (т. е. только с библиотекой C/C++), поэтому мы не можем импортировать ее должным образом в реду с борками Buildroot. Поэтому даже если вы собираете систему для ядра x86 или x86\_64, вам необязательно сгенерировать кросс-компиляционную инструментальную цепочку с помощью Buildroot или crosstool-NG.

Если вы хотите создать собственную цепочку инструментов для своего проекта, которую можно использовать как внешнюю цепочку инструментов в Buildroot, мы рекомендуем создать ее либо с помощью самого Buildroot (см. раздел [6.1.3](#)), либо с помощью [crosstool-NG](#).

Преимущества этого обэкэнда:

- Позволяет использовать известные и проверенные наборы инструментов кросс-компиляции.
- Позволяет избежать времени с борками цепочки инструментов кросс-компиляции, которое часто имеет большое значение в общем времени с борками встраиваемого приложения.

Недостатки этого обэкэнда:

- Если в вашем предварительно скомпилированном внешнем наборе инструментов есть ошибка, может быть сложно получить исправление от поставщика набора инструментов, если только вы не создадите свой внешний набор инструментов.

### 6.1.3 Создание внешней цепочки инструментов с помощью Buildroot

Однако внутренней цепочки инструментов Buildroot может использовать для создания внешней цепочки инструментов. Вот ряд шагов для создания внутренней цепочки инструментов и ее упаковки для повторного использования с самим Buildroot (или другими проектами).

Создайте новую конфигурацию Buildroot с следующими данными:

- Выберите соответствующие параметры Target для целевой архитектуры ЦП.
- В меню «Toolchain» оставьте значение по умолчанию Buildroot toolchain для типа Toolchain и настройте свой toolchain по своему усмотрению.
- В меню Конфигурации системы выберите None в качестве Init system и none в качестве /bin/sh
- В меню «Целевые пакеты» отключите BusyBox

- В меню «Образы файловой системы» отключите tar для корневой файловой системы.

Затем мы можем запустить сборку, а также попросить Buildroot сгенерировать SDK. Это удобно сгенерирует для нас tarball, который содержит нашу цепочку инструментов:

сделать sdk

Это создаст tar-архив SDK в \$(O)/images с именем, похожим на arm-buildroot-linux-uclibcgnueabi\_sdk-bui. Сохраните этот tar-архив, так как теперь это набор инструментов, который вы можете повторно использовать как внешний набор инструментов в других проектах Buildroot.

В других проектах Buildroot в меню Toolchain:

- Установите тип цепочки инструментов на Внешняя цепочка инструментов
- Установите для параметра «Toolchain» значение «Custom toolchain»
- Установите Toolchain origin на Toolchain для загрузки и установки
- Установите URL-адрес Toolchain на file:///path/to/your/sdk/tarball.tar.gz

#### 6.1.3.1 Внешняя оболочка цепочки инструментов

При использовании внешней цепочки инструментов Buildroot генерирует программу-обертку, которая прозрачно передает соответствующие параметры (в соответствии с конфигурацией) внешним программам цепочки инструментов. В случае, если вам нужно отладить эту обертку, чтобы проверить, какие именно arguments передаются, вы можете установить переменную окружения BR2\_DEBUG\_WRAPPER в одно из следующих значений:

- 0, если то или не установлено: нет отладки
- 1: трасировка всех arguments на одной строке
- 2: трасировка одного arguments на строку

## 6.2 /dev-управление

В системе Linux каталог /dev содержит специальные файлы, называемые файлами устройств, которые позволяют приложениям пользователям просматривать получать доступ к аппаратным устройствам, управляемым ядром Linux. Без этих файлов устройств ваши приложения пользовательским просмотра не могут использовать аппаратные устройства, даже если они правильно распознаются ядром Linux.

В разделе «Конфигурация системы», «Управление /dev» Buildroot предлагает четыре различных решения для работы с каталогом /dev:

- Первое решение — статическое с использованием таблицы устройств. Это старый классический способ обработки файлов устройств в Linux. При использовании этого метода файлы устройств постоянно ряжутся в корневой файловой системе (т. е. они сохраняются при перезагрузках), и нет ничего, что автоматически сдавалось бы и удаляло эти файлы устройств при добавлении или удалении аппаратных устройств из системы. Поэтому Buildroot сдается с стандартный набор файлов устройств с помощью таблицы устройств, причем файл по умолчанию хранится в system/device\_table\_dev.txt в исходном коде Buildroot. Этот файл обрабатывается как да Buildroot генерирует окончательный образ корневой файловой системы, поэтому файлы устройств не видны в вых одном целевом каталоге. Параметр BR2\_ROOTFS\_STATIC позволяет изменить таблицу устройств по умолчанию, используемую Buildroot, или добавить дополнительную таблицу устройств, чтобы Buildroot сдавал дополнительные файлы устройств во время сборки. Итак, если вы используете этот метод, файл устройства отсутствует в вашей системе, вы можете, например, создать файл board/<yourcompany>/<yourproject>/device\_table\_dev.txt, который содержит описание ваших дополнительных файлов устройств, а затем вы можете установить BR2\_ROOTFS\_STATIC\_DEVICE\_TABLE в system/device\_table\_dev.txt board/<yourcompany>/<yourproject>/device\_table\_dev.txt. Более подробную информацию о формате файла таблицы устройств см. в главе 25.

- Второе решение — динамическое, использующее только devtmpfs. devtmpfs — это виртуальная файловая система внутри ядра Linux, которая была введена в ядро 2.6.32 (если вы используете более старое ядро, использовать эту опцию невозможно). При монтировании в /dev эта виртуальная файловая система автоматически заставит файлы устройств появляться и исчезать по мере добавления и удаления аппаратных устройств из системы. Эта файловая система не сбрасывается при перезагрузках: она динамически заполняется ядром. Использование devtmpfs требует включения следующих параметров конфигурации ядра: CONFIG\_DEV TMPFS и CONFIG\_DEV TMPFS\_MOUNT. Когда Buildroot отвечает за сборку ядра Linux для вашего устройства, он обеспечивает включение этих двух параметров.

Однако если вы собираете ядро Linux вне Buildroot, то вы несете ответственность за включение этих двух опций (если вы этого не делаете, ваша система Buildroot не загрузится).

- Третье решение — динамическое с использованием devtmpfs + mdev. Этот метод также опирается на виртуальную файловую систему devtmpfs, подробно описанную выше (помимо требования иметь CONFIG\_DEV TMPFS и CONFIG\_DEV TMPFS\_MOUNT, включенные в конфигурацию ядра, по-прежнему применимы), но добавляет поверх нее утилиту пользователя для просмотра трансформации mdev. mdev — это программа-оболочка BusyBox, которую ядро будет вызывать каждый раз при добавлении или удалении устройства. Благодаря файлу конфигурации /etc/mdev.conf mdev можно настроить, например, для установки определенных разрешений или владельца файла устройства, вызова скрипта или приложения при появлении или исчезновении устройства и т. д. Помимо этого, он позволяет пользователю кому просматривать на события добавления и удаления устройства. mdev можно, например, использовать для автоматической загрузки модулей ядра при появлении устройства в системе. mdev также важен, если у вас есть устройство, которым требуется прошивка, так как он будет отвечать за передачу с одержимого прошивки в ядро. mdev — это облегченная реализация (меньшим количеством функций) udev. Более подробную информацию о mdev и его синтаксисе можно найти по адресу <http://git.busybox.net/busybox/tree/docs/mdev.txt>.
- Четвертое решение — динамическое с использованием devtmpfs + eudev. Этот метод также опирается на виртуальную файловую систему devtmpfs, подробно описанную выше, но добавляет поверх нее демон пользователя для просмотра трансформации eudev. eudev — это демон, работающий в фоновом режиме и вызываемый ядром при добавлении или удалении устройства из системы. Это более тяжелое решение, чем mdev, но обеспечивает большую гибкость. eudev — это автономная версия udev, оригинального демона пользователя для просмотра трансформации, используемого в большинстве дистрибутивов Linux для настольных компьютеров, который теперь является частью Systemd. Более подробную информацию см. на [сайте http://en.wikipedia.org/wiki/Udev](http://en.wikipedia.org/wiki/Udev).

Разработчики Buildroot рекомендуют начинать с динамического решения, использующего только devtmpfs, пока у вас не возникнет необходимости в уведомлении просмотра трансформации пользователя о добавлении/удалении устройства или не будет необходимости установки прошивок. В этом случае динамическое решение, использующее devtmpfs + mdev, обычно является ярко выраженным решением.

Обратите внимание, что если в качестве системы инициализации выбрана systemd, управление /dev будет осуществляться ядром udev, предоставляемой systemd.

## 6.3 Инициализация системы

Программа init — это первая программа пользователя для просмотра трансформации, запущенная ядром (она имеет номер PID 1) и отвечает за запуск служб и программ пользователя для просмотра трансформации (например, веб-сервера, графических приложений, других сетевых серверов и т. д.).

Buildroot позволяет использовать три различных типа систем инициализации, которые можно выбрать в разделе «Конфигурация с системами», «Система инициализации»:

- Первое решение — BusyBox. Среди многих программ BusyBox имеет реализацию базовой программы инициализации, которая доступна для большинства встраиваемых систем. Включение BR2\_INIT\_BUSYBOX гарантирует, что BusyBox сберегет и установит свою программу инициализации. Это решение по умолчанию в Buildroot. Программа инициализации BusyBox прочитает файл /etc/inittab при загрузке, чтобы знать, что делать. Синтаксис этого файла можно найти в <http://git.busybox.net/busybox/tree/examples/inittab> (обратите внимание, что синтаксис BusyBox inittab отличается: не используйте синтаксис в документации inittab из Интернета, чтобы узнать о BusyBox inittab). Inittab по умолчанию в Buildroot содержит явный package/busybox/inittab. Помимо монтирования нескольких файловых систем, новая задача inittab по умолчанию — это запуск криптооболочки /etc/init.d/rcS и запуск программы getty (которая обеспечивает приглашение для входа в систему).
- Второе решение — systemV. Это решение использует старую традиционную программу sysvinit, установленную в Buildroot в package/sysvinit. Это решение используется в большинстве настольных дистрибутивов Linux, пока они не перешли на более современные альтернативы, такие как Upstart или Systemd. sysvinit также работает с файлом inittab (синтаксис которого немного отличается от синтаксиса BusyBox). Файл inittab по умолчанию, устанавливаемый вместе с этим решением инициализации, находится в package/sysvinit/inittab.
- Третье решение — systemd. systemd — это система init нового поколения для Linux. Она делает гораздо больше, чем традиционные программы init: возможности агрессивного параллелизма, использует скрипты и активацию D-Bus для запуска служб, предлагает запуск демонов по требованию, отслеживает процессы с помощью контрольных групп Linux, поддерживает моментальные снимки и восстановление системы.

state и т. д. systemd будет полезен на относительно сложных системах, например, тех, которые требуют D-Bus и сервисов, взаимодействующих с ними. Стоит отметить, что systemd приносит довольно большое количество крупных зависимостей: dbus, udev и т. д. Более подробную информацию о systemd см. на <http://www.freedesktop.org/wiki/Software/systemd>.

Разработчики Buildroot рекомендуют использовать BusyBox init, так как это достаточно для большинства встраиваемых систем. В более сложных ситуациях можно использовать systemd.

## Глава 7

# Конфигурация других компонентов

Прежде чем пытаться изменить любой из перечисленных ниже компонентов, убедитесь, что вы уже настроили сам Buildroot и включили соответствующий пакет.

### BusyBox

Если у вас есть файл конфигурации BusyBox, вы можете напрямую указать этот файл в конфигурации Buildroot, ис пользуя `BR2_PACKAGE_BUSYBOX_CONFIG`. В противном случае Buildroot запускается с файла конфигурации BusyBox по умолчанию.

Для внесения последующих изменений в конфигурацию ис пользуйте `make busybox-menuconfig`, чтобы открыть редактор конфигурации BusyBox.

Также можно указать файл конфигурации BusyBox через переменную окружения `X` от этого не рекомендуется. Подробнее см. в разделе [8.6](#).

### uClibc

Настойка uClibc выполняется так же, как и для BusyBox. Переменная конфигурации для указания с помощью файла конфигурации — `BR2_UCLIBC_CONFIG`. Команда для внесения последующих изменений — `make uclibc-menuconfig`.

### Ядро Linux

Если у вас есть файл конфигурации ядра, вы можете напрямую указать этот файл в конфигурации Buildroot, ис пользуя `BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG`.

Если у вас еще нет файла конфигурации ядра, вы можете начать с указания `defconfig` в конфигурации Buildroot с помощью `BR2_LINUX_KERNEL_USE_DEFCONFIG` или начать с создания пустого файла и указания его как пользовательского файла конфигурации с помощью `BR2_LINUX_KERNEL_USE_CUSTOM_CONFIG`.

Для внесения последующих изменений в конфигурацию ис пользуйте команду `make linux-menuconfig`, чтобы открыть редактор конфигурации Linux.

### Barebox

Конфигурация Barebox выполняется так же, как и для ядра Linux. Соответствующие переменные конфигурации — `BR2_TARGET_BAREBOX_USE_CUSTOM_CONFIG` и `BR2_TARGET_BAREBOX_USE_DEFCONFIG`. Чтобы открыть редактор конфигурации, ис пользуйте `make barebox-menuconfig`.

### U-Boot

Настойка U-Boot (версии 2015.04 или более новой) выполняется так же, как и для ядра Linux. Соответствующие переменные конфигурации — `BR2_TARGET_UBOOT_USE_CUSTOM_CONFIG` и `BR2_TARGET_UBOOT_USE_DEFCONFIG`. Чтобы открыть редактор конфигурации, ис пользуйте `make uboot-menuconfig`.

# Глава 8

## Общее использование Buildroot

### 8.1 Сделать с советами

Это сборник советов, которые помогут вам максимально эффективно использовать Buildroot.

Отобразить все команды, выполненные make:

```
$ сделать V=1 <цель>
```

Отобразить список плат с defconfig:

```
$ сделать списокок плат с defconfigs
```

Показать все доступные цели:

```
$ сделать помощь
```

Не все цели всегда доступны, некоторые настройки в файле .config могут скрывать некоторые цели:

- busybox-menuconfig работает только при включенном busybox;
- linux-menuconfig и linux-savedefconfig работают только при включенном Linux;
- uclibc-menuconfig доступен только в том случае, если во внутреннем наборе инструментов выбрана библиотека uClibc C;
- barebox-menuconfig и barebox-savedefconfig работают только при включенном загрузчике barebox.
- uboot-menuconfig и uboot-savedefconfig работают только тогда, когда включен загрузчик U-Boot и uboot Система с борками настроена на Kconfig.

Очистка: Явная очистка требуется при изменении любых параметров конфигурации архитектуры или цепочки инструментов.

Чтобы удалить все продукты с борки (включая каталоги с борками, хост, промежуточные и целевые деревья образы и цепочку инструментов):

```
$ сделать чистым
```

Генерация руководства: Исподные тексты настоящего руководства находятся в каталоге docs/manual. Для генерации руководства:

```
$ сделать ручную очистку  
$ сделать руководство
```

Ручные исподные данные будут созданы в output/docs/manual.

ПРИМЕЧАНИЯ

- Для создания документации требуется несолько инструментов (см. раздел 2.2).

Сброс Buildroot для новой цели: Чтобы удалить все продукты с борки, а также конфигурацию:

```
$ сделать distclean
```

Примечание Если ccache включен, запуск make clean или distclean не очищает кеш компилятора, ис пользуемый Buildroot. Чтобы удалить его, обратитесь к разделу 8.13.3.

Сброс внутренних переменных make: можно сбросить переменные, известные make, вместе с их значениями:

```
$ make -s printvars VARS='ПЕРЕМЕННАЯ1 ПЕРЕМЕННАЯ2'
ПЕРЕМЕННАЯ1=значение_переменной
ПЕРЕМЕННАЯ2=значение_переменной
```

Вывод можно настроить с помощью некоторых переменных:

- VARS ограничивается списком переменными, имена которых соответствуют указанным шаблонам создания — это должно быть установлено, иначе ничего не будет выведено
- QUOTED\_VARS, если установлено значение YES, то значение будет заключено в одинарные кавычки
- RAW\_VARS, если установлено значение YES, будет выведено неразвернутое значение

Например:

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES
BUSYBOX_DEPENDENCIES=с келтнайц епока инс трументов
BUSYBOX_FINAL_ALL_DEPENDENCIES=с келтнайц епока инс трументов
BUSYBOX_FINAL_DEPENDENCIES=с келтнайц епока инс трументов
BUSYBOX_FINAL_PATCH_DEPENDENCIES=
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES QUOTED_VARS=YES
BUSYBOX_DEPENDENCIES='с келтнайц епока инс трументов'
BUSYBOX_FINAL_ALL_DEPENDENCIES='с келтнайц епока инс трументов'
BUSYBOX_FINAL_DEPENDENCIES='с келтнайц епока инс трументов'
BUSYBOX_FINAL_PATCH_DEPENDENCIES=''
BUSYBOX_RDEPENDENCIES='ncurses util-linux'
```

```
$ make -s printvars VARS=BUSYBOX_%DEPENDENCIES RAW_VARS=YES
BUSYBOX_DEPENDENCIES=с келтнайц епока инс трументов
BUSYBOX_FINAL_ALL_DEPENDENCIES=$(с ортировка $(BUSYBOX_FINAL_DEPENDENCIES) $(
    BUSYBOX_FINAL_PATCH_DEPENDENCIES))
BUSYBOX_FINAL_DEPENDENCIES=$(с ортировать $(BUSYBOX_DEPENDENCIES))
BUSYBOX_FINAL_PATCH_DEPENDENCIES=$(с ортировать $(BUSYBOX_PATCH_DEPENDENCIES))
BUSYBOX_RDEPENDENCIES=ncurses util-linux
```

Вывод переменных в кавычках можно повторно использовать в скриптах оболочки, например:

```
$ eval $(make -s printvars VARS=BUSYBOX_DEPENDENCIES QUOTED_VARS=YES) $ echo $BUSYBOX_DEPENDENCIES с келтнайц епока
инс трументов
```

## 8.2 Понимание того, когда необходи́ма полная перестройка

Buildroot не пытается определить, какие части системы должны быть перестроены при изменении конфигурации и системы через make menuconfig, make xconfig или один из других инструментов конфигурации. В некоторых случаях Buildroot должен перестроить всю систему, в некоторых случаях — только определенное подмножество пакетов. Но обнаружить это полностью надежным способом очень сложно, и поэтому разработчики Buildroot решили просто не пытаться это делать.

Вместо этого пользователь должен знать, когда необходимо полное восстановление. В качестве подсказки, вот несколько правил, которые помогут вам понять, как работать с Buildroot:

- При изменении конфигурации или целевой архитектуры требуется полная перестройка. Изменение варианта архитектуры, например, двоичный формат или стратегия плавающей точки оказывают влияние на всю систему.
- При изменении конфигурации toolchain обычно требуется полная перестройка. Изменение конфигурации toolchain часто влечет за собой изменение версии компилятора, типа библиотеки C или ее конфигурации или какого-либо другого фундаментального элемента конфигурации, и эти изменения оказывают влияние на всю систему.
- Когда в конфигурацию добавляется дополнительный пакет, полная перестройка не обязательно нужна. Buildroot определит, что этот пакет никогда не был собран, и сбера его. Однако, если этот пакет является библиотекой, которая может быть дополнительно использована пакетами, которые уже были собраны, Buildroot не будет автоматически пересобирать их. Либо вы знаете, какие пакеты должны быть пересобраны, и можете пересобрать их вручную, либо вам следует выполнить полную перестройку. Например, предположим, что вы сбрали систему с пакетом torrent, но без openssl. Ваша система работает, но вы понимаете, что отели бы иметь поддержку SSL в torrent, поэтому вы включаете пакет openssl в конфигурацию Buildroot и перезапускаете с боркой. Buildroot определит, что openssl должен быть собран, и сбера его, но он не определит, что torrent должен быть пересобран, чтобы воспользоваться ядром openssl для добавления поддержки OpenSSL. Вам придется либо выполнить полную перестройку, либо пересобрать сам torrent.
- Когда пакет удаляется из конфигурации, Buildroot не делает ничего обенного. Он не удаляет файлы, установленные этим пакетом из целевой корневой файловой системы или из цепочки инструментов sysroot. Для удаления этого пакета требуется полная перестройка борки. Однако, как правило, вам не обязательно удалять этот пакет прямо сейчас: вы можете подождать с ледяной паузой обрыва, чтобы перезапустить с боркой с нуля.
- При изменении подопечного пакета пакет не пересстраивается автоматически. После внесения таких изменений часто требуется достаточно перестроить только этот пакет, если только включение подопечного пакета не добавляет в пакет некоторые функции, полезные для другого пакета, который уже был собран. Отметим же, Buildroot не отслеживает, когда пакет должен быть пересобран: после сбоя пакета он никогда не пересстраивается, если только явно не указано делать это.
- При внесении изменений в скелет корневой файловой системы требуется полная перестройка. Однако при внесении изменений в оверлей корневой файловой системы, скрипт после сбоя или скрипт после образа нет необходиимости в полной перестройке: простой вызов make учитывает изменения.
- Когда пакет, указанный в FOO\_DEPENDENCIES, пересстраивается или удаляется, пакет foo не пересстраивается автоматически. Например, если пакет bar указан в FOO\_DEPENDENCIES с FOO\_DEPENDENCIES = bar и конфигурация пакета bar изменена, изменение конфигурации не приведет к автоматической перестройке пакета foo. В этом случае вам может потребоваться либо пересобрать все пакеты в вашей сборке, которые ссылаются на bar в своих DEPENDENCIES, либо выполнить полную перестройку, чтобы убедиться, что все зависит от bar пакеты обновлены.

В общем, если вы столкнулись с ошибкой с боркой и не уверены в возможных последствиях внесенных вами изменений конфигурации, выполните полную перестройку. Если вы получили ту же ошибку с боркой, то вы уверены, что ошибка связана с частичной перестройкой пакетов, и если эта ошибка возникает пакетами из официального Buildroot, не стесняйтесь сообщить о проблеме! По мере накопления опыта работы с Buildroot вы постепенно узнаете, когда действительно необходимо удалять пакет с боркой, и вы сэкономите много времени.

Для правки, полная перестройка достигается путем запуска:

```
$ сделать все чистым
```

## 8.3 Понимание того, как пересобирать пакеты

Один из наиболее частых вопросов, который задают пользователи Buildroot, — как пересобрать определенный пакет или как удалить пакет, не пересобирая все с нуля.

Удаление пакета не поддерживается Buildroot без пересборки с нуля. Это связано с тем, что Buildroot не отслеживает, какой пакет устанавливает какие файлы в каталогах output/staging и output/target, или какой пакет будет скомпилирован по-разному в зависимости от дистрибутива другого пакета.

Самый простой способ пересобрать отдельный пакет с нуля — удалить его каталог с борки в output/build. Затем Buildroot заново извлечет, заново настроит, заново скомпилирует и заново установит этот пакет с нуля. Вы можете попросить buildroot сделать это с помощью команды make <package>-dirclean.

С другой стороны, если вы хотите только перезапустить процесс сборки пакета с этапа его компиляции, вы можете запустить команду `make <package>-rebuild`. Он перезапустит компиляцию и установку пакета, но не с нуля он посредством повторно выполняет `make` и `make install` внутри пакета, поэтому он пересоберет только те файлы, которые изменились.

Если вы хотите перезапустить процесс сборки пакета с этапа его настройки, вы можете выполнить команду `make <package>-reconfigure`. Это перезапустит настройку, компиляцию и установку пакета.

В то время как `<package>-rebuild` подразумевает `<package>-reinstall`, а `<package>-reconfigure` подразумевает `<package>-rebu`, эти цели, а также `<package>` действуют только на указанный пакет и не вызывают пересоздание образа корневой файловой системы. Если необходимо пересоздание корневой файловой системы, следующий дополнительный запуск `make` или `make all`.

Внутри себя `Buildroot` создает так называемые файлы штампов, чтобы отслеживать, какие шаги в сборке были завершены для каждого пакета. Они хранятся в каталоге `output/build/<package>-<version>/` и называются `stamp_<step-name>`.

Описанные выше команды помогают манипулировать этими файлами штампов, чтобы заставить `Buildroot` перезапустить определенный набор шагов процесса сборки пакета.

Более подробная информация о целевых объектах с помощью пакета приведена в разделе [8.13.5](#).

## 8.4 Оффлайн сборки

Если вы собираетесь выполнить автономную сборку и просто хотите загрузить все исходные коды, которые вы ранее выбрали в конфигураторе (`menuconfig`, `nconfig`, `xconfig` или `gconfig`), то выполните:

```
$ создать ис точник
```

Теперь вы можете отключить или скопировать содержимое вашего каталога `all` на хост с сборкой.

## 8.5 Строительство из дерева

По умолчанию все, что создается `Buildroot`, сохраняется в каталоге `output` в дереве `Buildroot`.

`Buildroot` также поддерживает сборку из дерева с синтаксисом, подобным ядро Linux. Чтобы использовать его, добавьте `O=<directory>` в командную строку `make`:

```
$ make O=/tmp/build menuconfig
```

Или:

```
$ cd /tmp/build; make O=$PWD -C путь/к/корню /buildconfig
```

Все выходные файлы будут расположены в `/tmp/build`. Если путь `O` не существует, `Buildroot` его создаст.

Примечание: путь `O` может быть как абсолютным, так и относительным, но если он передается как относительный путь, важно отметить, что он интерпретируется относительно нового исходного каталога `Buildroot`, а не текущего рабочего каталога.

При использовании сборки из дерева, `Buildroot` .config и временные файлы также создаются в одном каталоге. Это означает, что вы можете безопасно запускать несколько сборок параллельно, используя одно и тоже исходное дерево, пока они используют уникальные выходные каталоги.

Для удобства использования `Buildroot` генерирует оболочку `Makefile` в выходном каталоге, поэтому после первого запуска вам больше не нужно передавать `O=<...>` и `-C <...>`, просто запустите (в выходном каталоге):

```
$ сделать <цель>
```

## 8.6 Переменные с рефами

Buildroot также учитывает некоторые переменные с рефами, когда они передаются в make или устанавливаются в рефе:

- HOSTCXX, хост-компилятор C++ для ис пользования
- HOSTCC, хост-компилятор C для ис пользования
- UCLIBC\_CONFIG\_FILE=<path/to/.config>, путь к файлу конфигурации uClibc, ис пользованому для компиляции uClibc, если сдается встроенный почка инструментов.

Обратите внимание, что файл конфигурации uClibc также можно настроить из интерфейса конфигурации, то есть через файл Buildroot.config; это рекомендуемый способ настройки.

- BUSYBOX\_CONFIG\_FILE=<path/to/.config>, путь к файлу конфигурации BusyBox.

Обратите внимание, что файл конфигурации BusyBox также можно настроить из интерфейса конфигурации, то есть через файл Buildroot.config; это рекомендуемый способ настройки.

- BR2\_CCACHE\_DIR для определения каталога, в котором Buildroot хранит кэшированные файлы при ис пользовании ccache.

- BR2\_DL\_DIR для определения каталога, в котором Buildroot хранит/извлекает загруженные файлы.

Обратите внимание, что каталог загрузки Buildroot также можно задать из интерфейса конфигурации, то есть через файл Buildroot.config. Подробнее о том, как задать каталог загрузки, см. в разделе [8.13.4](#).

- BR2\_GRAPH\_ALT, если установлено и не пусто, для ис пользования альтернативной цветовой схемы в графиках времени построения

- BR2\_GRAPH\_OUT для установки типа файла с создаваемых графиков: pdf (по умолчанию) или png.

- BR2\_GRAPH\_DEPS\_OPTS для передачи дополнительных параметров в график зависимости; см. раздел [8.9](#) для получения информации о принятых параметрах

- BR2\_GRAPH\_DOT\_OPTS передается ясновно в качестве параметров утилите dot длярисования графика зависимостей.

- BR2\_GRAPH\_SIZE\_OPTS для передачи дополнительных параметров в график размера; см. раздел [8.11](#) для получения информации о принятых параметрах

Пример, в котором ис пользованы файлы конфигурации, рас положенные в каталоге верхнего уровня в вашем \$HOME:

```
$ make UCLIBC_CONFIG_FILE=uClibc.config BUSYBOX_CONFIG_FILE=$HOME/bb.config
```

Если вы ис пользовали компилятор, отличный от gcc или g++, по умолчанию, для сборки и помогательных двоичных файлов на вашем хосте, то с делайте следующее:

```
$ с делай HOSTCXX=g++-4.3-HEAD HOSTCC=gcc-4.3-HEAD
```

## 8.7 Эффективная работа с образами файловой системы

Образы файловых систем могут быть довольно большими, в зависимости от выбранной вами файловой системы, количества пакетов, того, выделили ли вы с вободное пространство.. Однако некоторые места в образах файловых систем могут быть просто пустыми (например, длинная последовательность нулей); такой файл называется разреженным файлом.

Большинство инструментов могут эффективно обрабатывать разреженные файлы и будут сжимать или записывать только те части разреженного файла, которые не являются пустыми.

Например:

- tar принимает опцию -S, которая указывает ему хранить только не нулевые блоки разреженных файлов:

– tar cf archive.tar -S [files...] эффективно сжимает разреженные файлы в tarball – tar xf archive.tar -S эффективно сжимает

разреженные файлы, извлеченные из tarball

- cp принимает опцию --sparse=WHEN (WHEN может быть одним из значений auto, never или always):
  - cp --sparse=always source.file dest.file делает dest.file разреженным файлом, если source.file имеет длинные серии нулей

Другие инструменты могут иметь похожие опции. Пожалуйста, ознакомьтесь с соответствующими страницами руководства.

Вы можете использовать разреженные файлы, если вам необходимо скопировать образы файловой системы (например, для переноса с одного компьютера на другой) или если вам необходимо отправить их (например, команде вопросов и ответов).

Однако следует отметить, что пересылка образа файловой системы на устройство при использовании разреженного режима dd может привести к поломке файловой системы (например, может быть повреждена битовая карта блоков файловой системы ext2; или, если в вашей файловой системе есть разреженные файлы, эти части могут не быть полностью нулевыми при обратном чтении). Разреженные файлы следует использовать только при обработке файлов на машине с боркой, а не при передаче фактического устройства, которое будет использоваться янацели.

## 8.8 Подробная информация о пакетах

Buildroot может создать JSON-объявление, описывающее набор включенных пакетов в текущий конфигурации вместе с их зависимостями, лицензиями и другими метаданными. Это JSON-объявление создается с помощью show-info make target:

сделать show-инфо

Buildroot также может выдавать сведения о пакетах в виде HTML и JSON-вывода с помощью цели pkg-stats make. Среди прочего, эти сведения включают информацию о том, влияют ли известные CVE (увязимости безопасности) на пакеты в вашей текущей конфигурации. Он также показывает, есть ли более новая версия upstream для этих пакетов.

сделать pkg-stats тапистику

## 8.9 Графическое отображение зависимостей между пакетами

Одна из задач Buildroot — знать зависимости между пакетами и убедиться, что они собраны в правильном порядке. Иногда эти зависимости могут быть довольно сложными, и для данной системы часто бывает нелегко понять, почему тот или иной пакет был добавлен в сборку Buildroot.

Чтобы помочь понять зависимости, с ледовательно, лучше понять, какова роль различных компонентов во взаимодействии с системой Linux, Buildroot способен генерировать график зависимости.

Чтобы создать график зависимости всей скомпилированной вами системы, просто запустите:

сделать graph-зависим

Сгенерированный график вы найдете в файле output/graphs/graph-depends.pdf.

Если ваша система довольно большая, график зависимости может быть слишком большим и трудным для чтения. Поэтому можно сгенерировать график зависимости только для данного пакета

сделать <pkg>-график-зависим

Сгенерированный график вы найдете в output/graph/<pkg>-graph-depends.pdf.

Обратите внимание, что графики зависимости создаются с помощью инструмента dot из проекта Graphviz, который вы должны установить в своей системе для использования этой функции. В большинстве дистрибутивов он доступен как пакет graphviz.

По умолчанию графики зависимости генерируются в формате PDF. Однако, передав переменную среды BR2\_GRAPH\_OUT, вы можете переключиться на другие форматы вывода, такие как PNG, PostScript или SVG. Поддерживаются форматы, поддерживаемые опцией -T инструмента dot.

BR2\_GRAPH\_OUT=svg сделать graph-зависим

Поведение, зависящее от графика, можно контролировать, задавая параметры в переменной с реды BR2\_GRAPH\_DEPS\_OPTS.

Принимаются следующие варианты:

- --depth N, -d N, чтобы ограничить глубину зависимости до N уровней. Значение по умолчанию 0 означает отсутствие ограничений.
- --stop-on PKG, -s PKG, чтобы установить график на пакете PKG. PKG может быть фактическим именем пакета, глобальным символом, ключевым словом virtual (чтобы установить на виртуальных пакетах) или ключевым словом host (чтобы установить на основных пакетах). Пакет все еще присутствует в графике, но его зависимость — нет.
- --exclude PKG, -x PKG, как --stop-on, но также исключает PKG из графика.
- --transitive, --no-transitive, рисовать (или нет) транзитивные зависимости. По умолчанию транзитивные зависимости не рисуются
- --colors R,T,H, разделенный запятыми список цветов для рисования корневого пакета (R), цепелевых пакетов (T) и пакетов хоста (H). По умолчанию: lightblue, grey, gainsboro

```
BR2_GRAPH_DEPS_OPTS='--depth 3 --no-transitive --colors=red,green,blue' с делать график зависимостей
```

## 8.10 График продолжительности сборки

Когда сборка систмы занимает много времени, иногда полезно иметь возможность понять, какие пакеты собираются дольше всего, чтобы посмотреть, можно ли что-то сделать для укоренения сборки. Чтобы помочь такому анализу времени сборки, Buildroot собирает время сборки каждого шага каждого пакета и позволяет генерировать график из этих данных.

Чтобы сгенерировать график времени сборки после сборки, выполните:

сделать график-построить

Это сгенерирует набор файлов в output/graphs:

- build.hist-build.pdf, график рамма времени сборки для каждого пакета, упорядоченная в порядке сборки.
- build.hist-duration.pdf, график рамма времени сборки для каждого пакета, упорядоченная по длительности (сначала самые длинные)
- build.hist-name.pdf, график рамма времени сборки для каждого пакета, сортировка по имени пакета
- build.pie-packages.pdf, круговая диаграмма времени сборки для каждого пакета
- build.pie-steps.pdf, круговая диаграмма глобального времени, затраченного на каждый шаг процесса с сборки пакетов.

Для этой цели построения графиков требуется установление Python Matplotlib и NumPy (python-matplotlib и python-numpy в большинстве дистрибутивов), а также модуль argparse, если вы используете версию Python старше 2.7 (python-argparse в большинстве дистрибутивов).

По умолчанию формат для графика — PDF, но можно выбрать другой формат с помощью переменной окружения BR2\_GRAPH\_OUT. Единственный другой поддерживаемый формат — PNG:

```
BR2_GRAPH_OUT=png с создать график-построить
```

## 8.11 Графическое отображение вклада пакетов в размер файловой системы

Когда ваша целевая система работает, иногда полезно понять, какой вклад вносит каждый пакет Buildroot в общий размер корневой файловой системы. Чтобы помочь с таким анализом, Buildroot собирает данные о файлах, установленных каждым пакетом, и, используя эти данные, генерирует график и файлы CSV, детализирующие вклад размера различных пакетов.

Чтобы сгенерировать эти данные после сборки, выполните:

сделать размер графика

Это сгенерирует:

- output/graphs/graph-size.pdf, круговая диаграмма вклада каждого пакета в общий размер корневой файловой системы
- output/graphs/package-size-stats.csv, CSV-файл, показывающий вклад размера каждого пакета в общий размер корневой файловой системы
- output/graphs/file-size-stats.csv, CSV-файл, показывающий вклад размера каждого установленного файла в пакет, к которому он принадлежит и общему размеру файловой системы.

Для этого цели нужно создать ярус с установленной библиотекой Python Matplotlib (python-matplotlib в большинстве дистрибутивов), а также модуль argparse, если вы используете версию Python старше 2.7 (python-argparse в большинстве дистрибутивов).

Как и для графика продолжительности, переменная с реди BR2\_GRAPH\_OUT поддерживается для настройки формата выходных файлов. Подробнее об этой переменной с реди см. в разделе [8.9](#).

Кроме того, можно задать переменную окружения BR2\_GRAPH\_SIZE\_OPTS для дальнейшего управления сгенерированным графиком. Принимаются следующие параметры:

- --size-limit X, -l X, где группирует все пакеты, индивидуальный вклад которых ниже X процентов, в одну запись с меткой Others в графике. По умолчанию X=0.01, что означает, что пакеты, каждый из которых вносит менее 1%, группированы в Others. Допустимые значения находятся в диапазоне [0,0..1,0].
- --iec, --binary, --si, --decimal, для использования IEC (двоичная система с степенью 1024) или SI (десятичная система с степенью 1000; по умолчанию) префиксы.
- --biggest-first, чтобы сортировать пакеты в порядке убывания размера, а не в порядке возрастания размера.

**Примечание** Собранные данные о размере файловой системы имеют смысл только после полной перестройки. Обязательно запустите make clean all перед использованием make graph-size.

Чтобы сравнить размер корневой файловой системы двух разных компиляций Buildroot, например, после настройки конфигурации или при переключении на другой релиз Buildroot, используйте скрипт size-stats-compare. Он принимает два файла file-size-stats.csv (созданных make graph-size) в качестве входных данных. Более подробную информацию см. в правочном тексте этого скрипта:

утилиты/размер-статистика-сравнение -h

## 8.12 Параллельная сборка верхнего уровня

**Примечание.** В этом разделе рассматривается экспериментальная функция, которая, как известно, может работать без убытков в некоторых нестандартных ситуациях. Используйте ее на свой страх и риск.

Buildroot всегда может использовать параллельную сборку новых пакетов: каждый пакет собирается Buildroot с помощью make -jN (или эквивалентного вызова для систем сборки, не оснащенных make). Уровень параллелизма по умолчанию равен числу ЦП+1, но его можно настроить с помощью параметра конфигурации BR2\_JLEVEL.

Однако до 2020.02 Buildroot собирает пакеты последовательно: каждый пакет собирается один за другим, без параллелизации сборки между пакетами. Начиная с 2020.02 Buildroot имеет экспериментальную поддержку параллельной сборки верхнего уровня, что позволяет значительно экономить время сборки за счет параллельной сборки пакетов, не имеющих отношений зависимости. Однако эта функция отмечена как экспериментальная, как известно, в некоторых случаях не работает.

Чтобы использовать параллельную сборку верхнего уровня необходимо:

1. Включите опцию BR2\_PER\_PACKAGE\_DIRECTORIES в конфигурации Buildroot.

2. Используйте make -jN при запуске сборки Buildroot

Внутри BR2\_PER\_PACKAGE\_DIRECTORIES включите механизм, называемый каталогами для каждого пакета, который будет иметь следующие эффекты:

- Вместе с глобальным каталогом и глобальным каталогом host, общих для всех пакетов, будут использоваться левые и хостовые каталоги для каждого пакета, в \$(O)/per-package/<pkg>/target/ и \$(O)/per-package/<pkg>/host/ соответственно. Эти папки будут заполнены из соответствующих папок зависимостей пакета в начале с борки <pkg>. Таким образом, компилятор и все другие инструменты смогут видеть и получать доступ только к тем файлам, которые установлены зависимостями, явно перечисленными в <pkg>.
- В конце с борки будут заполнены глобальные каталоги target и host, расположенные в \$(O)/target и \$(O)/host соответственно. Это означает, что во время с борки эти папки будут пустыми, и только в самом конце с борки они будут заполнены.

## 8.13 Расширенное использование

### 8.13.1 Использование сгенерированной цепочки инструментов вне Buildroot

Вы можете захотеть скомпилировать для своей цели с обстановочные программы или другое программное обеспечение, которое не упаковано в Buildroot. Для этого вы можете использовать цепочку инструментов, сгенерированную Buildroot.

Инструментальная цепочка, сгенерированная Buildroot, по умолчанию находится в output/host/. Самый простой способ ее использования — добавить output/host/bin в переменную среды PATH, а затем использовать ARCH-linux-gcc, ARCH-linux-objdump, ARCH-linux-ld и т. д.

В качестве альтернативы Buildroot может также экспортить цепочку инструментов и файлы разработки всех выбранных пакетов в качестве SDK, выполнив команду make sdk. Это сгенерирует tarball с содержимым каталога host output/host/, названный <TARGET-TUPLE>\_S (который можно определить, установив переменную окружения BR2\_SDK\_PREFIX) и расположенный в каталоге вывода output/image.

Затем этот tarball можно распаковать с реди разработчиков приложений, когда они захотят разработать свои приложения, которые (еще) не упакованы в пакет Buildroot.

После извлечения архива SDK пользователь должен запустить скрипт relocate-sdk.sh (расположенный в верхнем каталоге SDK), чтобы убедиться что все пути обновлены с учетом нового расположения.

В качестве альтернативы, если вы просто хотите подготовить SDK без генерации tarball (например, потому что вы просто перемещаете каталог host или создаете tarball самостоятельно), Buildroot также позволяет вам просто подготовить SDK с помощью make prepare-sdk без фактической генерации tarball.

Для вашего удобства, выбрав опцию BR2\_PACKAGE\_HOST\_ENVIRONMENT\_SETUP, вы можете получить скрипт environment-setup, установленный в output/host/ и, следовательно, в вашем SDK. Этот скрипт может быть получен с помощью .your/sdk/path/environment для каждого порта ряда переменных среды, которые помогут вам скомпилировать ваши проекты с помощью Buildroot SDK: PATH будет содержать двайчные файлы SDK, стандартные переменные autotools будут определены с соответствующими значениями, а CONFIGURE\_FLAGS будет содержать новые параметры ./configure для каждого скомпилируемого проекта autotools. Он также предоставляет несколько полезных команд. Однако обратите внимание, что после получения этого скрипта рекомендуется использовать его только для каждого скомпилируемого проекта, а не для обстановки компиляции.

### 8.13.2 Использование gdb в Buildroot

Buildroot позволяет выполнять отладку, при которой отладчик запускается на машине с борки и взаимодействует с gdbserver на левой машине для управления выполнением программы.

Чтобы это сделать:

- Если вы используете внутреннюю цепочку инструментов (создана Buildroot), вы должны включить BR2\_PACKAGE\_HOST\_GDB, BR2\_PACKAGE\_GDB и BR2\_PACKAGE\_GDB\_SERVER. Это гарантирует, что и отладка с -gdb, и gdbserver будут собраны, и что gdbserver будет установлен на вашей цели.
- Если вы используете внешнюю цепочку инструментов, вам следует включить BR2\_TOOLCHAIN\_EXTERNAL\_GDB\_SERVER\_COPY, который скопирует gdbserver, включенный во внешнюю цепочку инструментов, в цель. Если ваша внешняя цепочка инструментов не имеет отладки с -gdb или gdbserver, также можно позволить Buildroot построить их, включив те же параметры, что и для внутренней обстановки цепочки инструментов.

Теперь, чтобы начать отладку программы с именем foo, вам следует запустить на цели:

```
gdbserver :2345 foo
```

Это заставит gdbserver прослушивать TCP-порт 2345 на предмет соединений от кросс-гудб.

Затем на хосте следует запустить кросс-ГУДБ с помощью следующей командной строки:

```
<buildroot>/output/host/bin/<корреж>-gdb -ix <buildroot>/output/staging/usr/share/buildroot/gdbinit foo
```

Конечно, foo должен быть доступен в текущем каталоге, сопрограммой с отладочными символами. Обычно вы запускаете эту команду из каталога, где содержится foo (а не из output/target/, поскольку двоичные файлы в этом каталоге обрезаются).

Файл <buildroot>/output/staging/usr/share/buildroot/gdbinit сообщает кросс-гудб, где найти библиотеки цели.

Наконец, чтобы подключиться к цели из cross gdb:

```
(gdb) target remote <целевой IP-адрес>:2345
```

### 8.13.3 Использование ccache в Buildroot

**ccache** это кеш компилятора. Онхранит объектные файлы, полученные в результате каждого процесса с компиляцией, и может пропускать будущую компиляцию тогоже исходного файла (с тем же компилятором и теми же аргументами), используя уже существующие объектные файлы. При выполнении практических идентичных сборок с нуля несколько раз, он может значительно ускорить процесс сборки. Поддержка ccache

интегрирована в Buildroot. Вам просто нужно включить кеш компилятора в параметрах сборки. Это автоматически сконфигурирует ccache и будет использовать его для каждой хостовой и целевой компиляции.

Кеш находится в каталоге, определенном параметром конфигурации BR2\_CCACHE\_DIR, который по умолчанию равен \$HOME/.buildroot. Это место расположения по умолчанию находится за пределами выходного каталога Buildroot, поэтому его можно сместить с помощью отдельными сборками Buildroot. Если вы хотите избавиться от кеша, просто удалите этот каталог.

Вы можете получить статистику по кешу (его размер, количество попаданий, промахов и т. д.), запустив make ccache-stats.

make target ccache-options и переменная CCACHE\_OPTIONS предстаивают более общий доступ к ccache. Например

```
# установить предельный размер кеша
сделать CCACHE_OPTIONS="--max-size=5G" ccache-options

# счетчики нулевой статистики делаются
CCACHE_OPTIONS="--zero-stats" ccache-options
```

ccache создает эшиды одних файлов и опций компилятора. Если опция компилятора отличается, кэшированный объектный файл не будет использоваться. Однако многие опции компилятора с одержат абсолютный путь к промежуточному каталогу. Из-за этого сборка в другом выходном каталоге приведет к множеству промахов кеша.

Чтобы избежать этой проблемы, buildroot имеет опцию Use relative paths (BR2\_CCACHE\_USE\_BASEDIR). Это перепишет все абсолютные пути, которые указывают внутрь выходного каталога, в относительные пути. Таким образом, изменение выходного каталога больше не приводит к промахам кеша.

Недостатком относительных путей является то, что они также оказываются относительными путями в объектном файле. Поэтому, например, отладчик больше не найдет файл, если вы сначала не перейдете в выходной каталог.

См. раздел руководства ccache «Компиляция в разных каталогах», для получения более подробной информации об этой перезаписи абсолютных путей.

Когда ccache включен в Buildroot с помощью параметра BR2\_CCACHE=y:

- ccache используется во время сборки Buildroot
- ccache не используется при сборке вне Buildroot, например, при прямом вызове кросс-компилятора или исользовании SDK

Это поведение можно переопределить с помощью переменной среды BR2\_USE\_CCACHE: если установлено значение 1, использование ccache включено (значение по умолчанию во время сборки Buildroot), если не установлено или установлено значение, отличное от 1, использование ccache отключено.

### 8.13.4 Расположение загружаемых пакетов

Различные tarballs, которые загружаются в Buildroot, хранятся в BR2\_DL\_DIR, который по умолчанию является каталогом dl. Если вы хотите создать полную версию Buildroot, которая как известно, работает с соответствующими tarballs, вы можете сделать копию этого каталога. Это позволит вам повторно генерировать toolchain и целевую файловую систему с точно такими же версиями.

Если вы поддерживаете несколько деревьев в Buildroot, может быть лучше иметь общее место загрузки. Этого можно добиться указав переменную с редью BR2\_DL\_DIR на каталог . Если это установлено, то значение BR2\_DL\_DIR в конфигурации Buildroot определяется. Следующая строка должна быть добавлена в <~/bashrc>.

```
export BR2_DL_DIR=<общее место загрузки>
```

Место загрузки также можно задать в файле .config с помощью параметра BR2\_DL\_DIR. В отличие от большинства параметров в файле .config, это значение определяется языковой переменной с редью BR2\_DL\_DIR.

### 8.13.5 Цели make, специфичные для пакета

Запуск команды make <package> собирает и устанавливает данный пакет и его зависимости.

Для пакетов, использующих инфраструктуру Buildroot, существует множество специальных целей make, которые можно вызывать независимо, например:

```
сделать <пакет><цель>
```

Цели с борками пакета (в порядке их выполнения):

Описание команды/цели	
источник	Получите исходный код (загрузите tarball, клонируйте исходный репозиторий и т. д.)
зависит от	Собрать и установить все зависимости, необязательные для сборки пакета.
извлекать	Поместите исходный код в каталог с борками пакета (извлеките tarball, скопируйте исходный код и т. д.)
патч	Примените запatches, если такие имеются
настроить с боркой	Запустите команды настройки, если такие имеются
	Запустите команды компиляции install-
staging target package: Запустите установку пакета в промежуточном каталоге, если необязательный	
целевой пакет install-target: Запустите установку пакета в целевом каталоге, если необязательный	
установить	целевой пакет: выполните 2 предыдущие команды установки пакета xоста запустите установку пакета в каталоге xоста

Кроме того, есть еще несколько полезных целей make:

команда/цель	Описание
show-dependencies	Отображает зависимости первого порядка, необязательные для сборки пакета show-recursive-dependencies Рекурсивно отображает зависимости, необязательные для сборки пакета show-rdepends Отображает обратные зависимости первого порядка пакета (т. е. пакеты, которые напрямую зависят от этого)
show-recursive-rdepends	Рекурсивно отображает обратные зависимости пакета (т. е. пакеты, которые зависят от этого, прямо или косвенно)
graph-dependencies Генерирует граф зависимостей пакета в контексте текущего Buildroot Конфигурация Подробнее о графиках зависимостей см. в этом разделе .	
graph-rdepends Создать график обратных зависимостей этого пакета (т. е. пакетов, которые зависят от него, прямо или косвенно)	
graph-both-dependencies Создать график этого пакета в обоих направлениях (т. е. пакеты, которые зависят от него и от которого он зависит, прямо или косвенно)	
dirclean	Удалить весь каталог с борками пакета
пересобранить	Повторно выполните команды установки.

Команда/цель	Описание
	Повторно выполните команды компиляции — это имеет смысл только при использовании функции OVERRIDE_SRCDIR или если вы изменили файл непосредственно в каталоге с борками, выполните
повторную нас тройку. Повторно выполните команды нас тройки, затем выполните повторную с боркой — это имеет смысл только при использовании функции OVERRIDE_SRCDIR или когда вы изменили файл непосредственно в каталоге с борками	

### 8.13.6 Использование Buildroot во время разработки

Обычная операция Buildroot заключается в загрузке tarball, его извлечении, нас тройке, компиляции и установке программного компонента, находящегося внутри этого tarball. Используя однократный код извлекается в output/build/<package>-<version>, который является временным каталогом всякий раз, когда используется make clean, этот каталог полностью удаляется и создается заново при следующем вызове make. Даже когда в качестве всех однократных данных для каждого пакета используется репозиторий Git или Subversion, Buildroot сдвигает из него tarball, а затем ведет с багажем, как обычно делает tarball.

Такое поведение хорошо подходит, когда Buildroot используется в основном как инструмент интеграции, для сборки и интеграции всех компонентов встроенной системы Linux. Однако, если вы используете Buildroot во время разработки определенных компонентов системы, такое поведение не очень удобно вместе с этим вы можете внести небольшое изменение в исходный код одного пакета и иметь возможность быстро пересобрать систему с помощью Buildroot.

Внесение изменений непосредственно в output/build/<package>-<version> не является подходящим решением, поскольку этот каталог удаляется при выполнении make clean.

Поэтому Buildroot предоставляет специальный механизм для этого варианта использования механизма <pkg>\_OVERRIDE\_SRCDIR. Buildroot считывает файл переопределения, который позволяет пользователю сообщить Buildroot местоположение источника для определенных пакетов.

Расположение файла переопределения по умолчанию — \$(CONFIG\_DIR)/local.mk, как определено параметром конфигурации BR2\_PACKAGE\_OVERRIDE\_FILE. \$(CONFIG\_DIR) — это расположение файла Buildroot .config, поэтому local.mk по умолчанию находится рядом с файлом .config, что означает:

- В каждом каталоге Buildroot верхнего уровня для сборки дерева (т.е. когда O= не используется)
- В каталоге выше дерева для сборки дерева (т.е. когда используется я O=)

Если требуется явное расположение, чем указано по умолчанию, его можно указать с помощью параметра конфигурации BR2\_PACKAGE\_OVERRIDE\_FILE.

В этом файле переопределения Buildroot ожидает найти строки следующего вида:

```
<pkg1>_OVERRIDE_SRCDIR = /путь/к/pkg1/источникам
<pkg2>_OVERRIDE_SRCDIR = /путь/к/pkg2/источникам
```

Например:

```
LINUX_OVERRIDE_SRCDIR = /home/bob/linux/
BUSYBOX_OVERRIDE_SRCDIR = /home/bob/busybox/
```

Когда Buildroot обнаруживает, что для данного пакета определен <pkg>\_OVERRIDE\_SRCDIR, он больше не будет пытаться загружать, извлекать и исправлять пакет. Вместо этого он будет напрямую использовать исходный код, доступный в указанном каталоге, и make clean не будет трогать этот каталог. Это позволяет указать Buildroot на ваши собственные каталоги, которыми можно управлять с помощью Git, Subversion или любой другой системы контроля версий. Для этого Buildroot будет использовать rsync для копирования исходного кода компонента из указанного <pkg>\_OVERRIDE\_SRCDIR в output/build/<package>-custom/.

Этот механизм лучше всего использовать с целями make <pkg>-rebuild и make <pkg>-reconfigure. Последовательность make <pkg>-rebuild all выполнит rsync исходного кода из <pkg>\_OVERRIDE\_SRCDIR в output/build/<рас (благодаря rsync копируются только измененные файлы) и перезапустит процесс с борками только этого пакета.

В приведенном выше примере пакета Linux разработчик может затем внести изменения в исходный код в /home/bob/linux, а затем запустить:

```
с делать linux-rebuild все
```

и в считанные секунды получает обновленный образ ядра Linux в output/images. Аналогично можно внести изменения исходный код BusyBox в /home/bob/busybox, и после:

```
с делать busybox-перестроить все
```

Образ корневой файловой системы в output/images содержит обновленный BusyBox.

Исходные деревья для больших проектов часто содержат отниты и лишние файлы, которые не нужны для сборки, но замедляют процесс копирования исходников с помощью rsync. При желании можно определить <pkg>\_ OVERRIDE\_SRCDIR\_RSYNC\_EXCLUSIONS, чтобы пропустить с них ронизацию определенных файлов из исходного дерева. Например, при работе с пакетом webkitgtk следующее исключит тесты и сборки в дереве из локального исходного дерева WebKit:

```
WEBKITGTK_OVERRIDE_SRCDIR = /home/bob/WebKit
WEBKITGTK_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS = \
    --exclude JSTests --exclude ManualTests --exclude PerformanceTests \
    --exclude WebDriverTests --exclude WebKitBuild --exclude WebKitLibraries \
    --exclude WebKit.xcworkspace --exclude Веб-сайты --exclude Примеры
```

По умолчанию Buildroot пропускает с них ронизацию артефактов VCS (например, каталоги .git и .svn). Некоторые пакеты предпочитают иметь эти каталоги VCS доступными во время сборки, например, для автоматического определения точной ссылки на коммит для информации о версии.

Чтобы отменить всестроенную фильтрацию с ними склонениями, добавьте обратно эти каталоги:

```
LINUX_OVERRIDE_SRCDIR_RSYNC_EXCLUSIONS = --include .git
```

# Глава 9

## Индивидуальная настройка под конкретный проект

Типичные действия, которые вам может потребоваться выполнить для конкретного проекта:

- Настройка Buildroot (включая параметры сборки и набор инструментов, выбор типа образа загрузчика, ядра, пакета и файловой системы)
- Настройка других компонентов, таких как ядро Linux и BusyBox
- Настройка сгенерированной целевой файловой системы
  - добавление или перезапись файлов в целевой файловой системе (использование BR2\_ROOTFS\_OVERLAY) – изменение или удаление файлов в целевой файловой системе (использование BR2\_ROOTFS\_POST\_BUILD\_SCRIPT) – запуск произвольных команд перед созданием образа файловой системы (использование BR2\_ROOTFS\_POST\_BUILD\_SCRIPT) – настройка прав доступа к файлам и владельца (использование BR2\_ROOTFS\_DEVICE\_TABLE) – добавление узлов пользовательских устройств (использование BR2\_ROOTFS\_STATIC\_DEVICE\_TABLE)
- Добавление пользовательских учетных записей (использование BR2\_ROOTFS\_USERS\_TABLES)
- Запуск произвольных команд после создания образа файловой системы (с использованием BR2\_ROOTFS\_POST\_IMAGE\_SCRIPT)
- Добавление патчей для конкретного проекта в некоторые пакеты (использование BR2\_GLOBAL\_PATCH\_DIR)
- Добавление пакетов, специфичных для проекта

Важное замечание относительно таких настроек, специфичных для проекта: пожалуйста, внимательно рассмотрите, какие изменения действительно специфичны для проекта, а какие изменения также полезны разработчикам за пределами вашего проекта. С сообщество Buildroot настоятельно рекомендует и поощряет передачу улучшений, пакетов и поддержки платы в официальный проект Buildroot. Конечно, иногда невозможно или нежелательно передавать, поскольку изменения являются ячейкой с специфичными или proprietарными.

В этой главе описывается, как создавать такие специфичные для проекта настройки в Buildroot и как их ранить таким образом, чтобы вы могли использовать тот же образ воспроизводимым образом, даже после запуска make clean. Следуя рекомендации с трети, вы даже можете использовать один и тот же дерево Buildroot для нескольких различных отдельных проектов!

### 9.1 Рекомендуемая структура каталогов

При настройке Buildroot для вашего проекта вы создадите один или несколько файлов, специфичных для проекта, которые необходимо однаждо склонировать. Отметьте, что большинство этих файлов можно разместить в любом месте, поскольку их путь должен быть указан в конфигурации Buildroot, разработчики Buildroot рекомендуют определенную структуру каталогов, которая описана в этом разделе.

Организацию этой структуры каталогов вы можете выбрать, где разместить саму эту структуру: либо внутри дерева Buildroot, либо вне его, используя дерево br2-external. Оба варианта допустимы, выберите тот, который вам больше нравится.

```

+-- доска/
|   +-- <компания>/
|       +-- <имя_доски>/
|           +-- linux.config
|           +-- busybox.config
|           +-- <другие файлы конфигурации>
|           +-- post_build.sh
|           +-- post_image.sh
|           +-- rootfs_overlay/
|               | +-- ит.д./
|               | +-- <некоторые файлы>
|               +-- патчи/
|                   +-- фу/
|                       | +-- <некоторые патчи>
|                   +-- libbar/
|                       +-- <некоторые другие патчи>

+-- конфиги/
|   +-- <имя_платы>_defconfig
|
+-- пакет/
|   +-- <компания>/
|       +-- Config.in (если не используется внешнее дерево br2)
|       +-- <company>.mk (если не используется внешнее дерево br2)
|           +-- пакет1/
|               |       +-- Конфигурация од
|               |       +-- package1.mk
|           +-- пакет2/
|               |       +-- Конфигурация од
|               |       +-- package2.mk
|
+-- Config.in (если используется внешнее дерево br2)
+-- external.mk (если используется дерево br2-external)
+-- external.desc (если используется дерево br2-external)

```

Подробная информация о файлах, показанных выше, приведена далее в этой главе.

Примечание: если вы решили разместить эту структуру вне дерева Buildroot, но в дереве br2-external, то <company> и, возможно, компоненты <boardname> могут быть излишними и их можно опустить.

### 9.1.1 Реализация множественных настроек

Довольно часто пользователь имеет несколько связанных проектов, которые частично требуют одинаковых настроек. Вместе с тем, чтобы дублировать для этих настроек для каждого проекта рекомендуется использовать многоуровневый подход к настройке, как описано в этом разделе.

Почти все методы настройки, доступные в Buildroot, такие как скрипты после сборки и наложения корневой файловой системы, принимают список элементов, разделенных пробелами. Указанные элементы всегда обрабатываются по порядку, слева направо. При создании более одного такого элемента, один для общих настроек и другой для действительно специфичных для проекта настроек, вы можете избежать ненужное дублирование. Каждый слой обычно включает отдельным каталогом внутри board/<company>/. В зависимости от своих проектах вы можете даже ввести более двух слоев.

Пример структуры каталогов, где у пользователя есть два уровня настроек: common и fooboard:

```

+-- доска/
    +-- <компания>/
        +-- общий/
            | +-- post_build.sh
            | +-- rootfs_overlay/
            | | +-- | +-- патчи/ ...
            |
            |       +-- ...

```

```

| +- fooboard/
  +- linux.config +-
  busybox.config +- <другие
  файлы конфигурации> +- post_build.sh +-
  rootfs_overlay/ | +- +-
  patches/
  ...
  +- ...

```

Например, если у пользователя установлен параметр конфигурации BR2\_GLOBAL\_PATCH\_DIR следующим образом:

```
BR2_GLOBAL_PATCH_DIR="board/<company>/common/patches board/<company>/fooboard/patches"
```

затем сначала будут применены патчи из общего слоя, а затем патчи из слоя fooboard.

## 9.2 Установка настроек вне Buildroot

Как уже кратко упоминалось в разделе [9.1](#), вы можете разместить специфичные для проекта настройки в двух местах:

- непосредственно в дереве Buildroot, обычно поддерживая их с помощью ветвей в системе контроля версий, чтобы обновление до более новую версию Buildroot получить легко.
- вне дерева Buildroot, используя механизм br2-external. Этот механизм позволяет устанавливать рецели пакетов, поддержку платы и файлы конфигурации вне дерева Buildroot, при этом сюда вносятся изменения в логику сборки. Мы называем это расположение деревом br2-external. В этом разделе объясняется, как использовать механизм br2-external и что следует предстартовать в дереве br2-external.

Можно указывать Buildroot использовать одно или несколько деревьев br2-external, установив переменную make BR2\_EXTERNAL в пути(и) к дереву(ям) br2-external, которое нужно использовать. Ее можно передать в любой вызов Buildroot make. Она автоматически сюда вносится с кратким файлом .br2-external.mk в видах одном каталоге. Благодаря этому нет необходимости передавать BR2\_EXTERNAL при каждом вызове make. Однако ее можно изменить в любое время передав новое значение, и удалить, передав пустое значение.

Примечание. Путь к внешнему дереву br2 может быть как абсолютным, так и относительным. Если он передается как относительный путь, важно отметить, что он интерпретируется относительно нового исходного каталога Buildroot, а не вида одного каталога Buildroot.

Примечание: Если вы используете дерево br2-external из версии Buildroot 2016.11, вам необходимо преобразовать его, прежде чем вы сможете использовать его с Buildroot 2016.11 и выше. См. раздел [27.2](#) для получения правки о том, как это сделать.

Вот несколько примеров:

```
buildroot/ $ make BR2_EXTERNAL=/путь/к/меню_config
```

С этого момента будут использоваться определения из дерева /path/to/foo br2-external:

```
buildroot/ $ make buildroot/
$ make юридическая информация
```

Мы можем переключиться на другое дерево br2-external в любой момент:

```
buildroot/ $ make BR2_EXTERNAL=/где/у/нас/есть/bar/xconfig
```

Мы также можем использовать несколько внешних деревьев br2:

```
buildroot/ $ make BR2_EXTERNAL=/путь/к/foo:/где/у/нас/есть/bar menuconfig
```

Или отключите использование любого внешнего дерева br2:

```
buildroot/ $ make BR2_EXTERNAL= xconfig
```

## 9.2.1 Макет внешнего дерева br2

Дерево br2-external должно содержать как минимум три файла, описанных в следующих главах:

- внешний.desc
- внешний.mk
- Конфигурация

Помимо этих обязательных файлов, может быть дополнительный и необязательный контент, который может присутствовать в дереве br2-external, например, каталог configs/ или provide/. Они также описаны в следующих главах.

Полный пример внешнего дерева br2 также описан ниже.

### 9.2.1.1 Файл external.desc

Этот файл описывает дерево br2-external: имя и описание этого дерева br2-external.

Формат этого файла основан на строках, каждая строка начинается с ключевого слова, за которым следует двоеточие и один или несколько пробелов, за которыми следует значение, присвоенное этому ключевому слову. В настоящем времени распознаются два ключевых слова:

- name, обязательно, определяет имя для этого дерева br2-external. Это имя должно использовать только символы ASCII в наборе [A-Za-z0-9\_]; любые другие символы запрещены. Buildroot устанавливает переменную BR2\_EXTERNAL\_\${NAME}\_PATH на абсолютный путь дерева br2-external, чтобы вы могли использовать ее для ссылки на ваше дерево br2-external. Эта переменная доступна как в Kconfig, так и в Makefile, чтобы вы могли использовать ее для включения других Makefile (см. ниже) или ссылки на другие файлы (например, файлы данных) из вашего дерева br2-external.

**Примечание:** Поскольку можно использовать несколько деревьев в br2-external одновременно, это имя используется Buildroot для генерации переменных для каждого из этих деревьев. Это имя используется для идентификации вашего дерева br2-external, поэтому постарайтесь придумать имя, которое действительно описывает ваше дерево br2-external, чтобы оно было действительно уникальным, и не конфликтовало с другим именем из другого дерева br2-external, особенно если вы планируете каким-то образом поделиться им с вашим деревом br2-external с третьими лицами или использовать деревья br2-external от третьих лиц.

- desc, необязательно, предоставляет краткое описание для этого дерева br2-external. Оно должно умещаться на одной строке, в основном имеет свободную форму (см. ниже) и используется при отображении информации о дереве br2-external (например, на списке файлов defconfig или в качестве подсказки в menupconfig); как такое, оно должно быть относительно кратким (вероятно, 40 символов — хороший верхний предел). Описание доступно в переменной BR2\_EXTERNAL\_\${NAME}\_DESC.

Примеры имен и соответствующих переменных BR2\_EXTERNAL\_\${NAME}\_PATH:

- FOO BR2\_EXTERNAL\_FOO\_PATH
- BAR\_42 BR2\_EXTERNAL\_BAR\_42\_PATH

В следующих примерах предполагается, что имя установлено на BAR\_42.

**Примечание:** Оба BR2\_EXTERNAL\_\${NAME}\_PATH и BR2\_EXTERNAL\_\${NAME}\_DESC доступны в файлах Kconfig и Makefiles. Они также экспортируются в среду, поэтому доступны в скриптах post-build, post-image и in-fakeroott.

### 9.2.1.2 Файлы Config.in и external.mk

Эти файлы (каждый из которых может быть пустым) можно использовать для определения рецептов пакетов (например, foo/Config.in и foo/foo.mk, как для пакетов, объединенных в сам Buildroot) или других пользовательских параметров конфигурации или логики make.

Buildroot автоматически включает Config.in из каждого дерева br2-external, чтобы он отображался в меню конфигурации верхнего уровня, а также включает external.mk из каждого дерева br2-external с остальной логикой makefile.

Основное использование этого — ранение рецептов пакетов. Рекомендуемый способ сделать это — написать файл Config.in, который выглядит так:

ис точник "\$BR2\_EXTERNAL\_BAR\_42\_PATH/package/package1/Config.in" ис точник  
"\$BR2\_EXTERNAL\_BAR\_42\_PATH/package/package2/Config.in"

Затем создайте файл external.mk, который выглядит следующим образом:

```
включить $(сорттировать $(wildcard $(BR2_EXTERNAL_BAR_42_PATH)/package/*/*.mk))
```

А затем в \$(BR2\_EXTERNAL\_BAR\_42\_PATH)/package/package1 и \$(BR2\_EXTERNAL\_BAR\_42\_PATH)/package с создайте обычные рецепты пакетов Buildroot, как описано в Главе 18. При желании вы также можете сгруппировать пакеты в подкаталоги с именем <boardname> и соответствующим образом адаптировать указанные выше пути.

Вы также можете определить пользовательские параметры конфигурации в Config.in и пользовательскую логику с созданиями в external.mk.

### 9.2.1.3 Каталог configs/

Можно хранить defconfigs Buildroot в подкаталоге configs дерева br2-external. Buildroot автоматически покажет их в выводе make list-defconfigs и позволит загружать их с помощью обычной команды make <name>\_defconfig. Они будут видны в выводе make list-defconfigs, под меткой External configs, с содержащей имя дерева br2-external, в котором они определены.

**Примечание:** Если файл defconfig присутствует в более чем одном дереве br2-external, то ис пользуется файл из последнего дерева br2-external. Таким образом, можно переопределить defconfig, упакованный в Buildroot или другое дерево br2-external.

### 9.2.1.4 Каталог provide/

Для некоторых пакетов Buildroot предоставляет выбор между двумя (или более) реализациями API-совместимых таких пакетов. Например, есть выбор между libjpeg или jpeg-turbo; есть выбор между openssl или libressl; есть выбор между известными, предварительно настроеными цепочками инструментов...

br2-external может расширять эти возможности, предоставив набор файлов, определяющих эти альтернативы:

- provide/toolchains.in определяет предварительно настроенные цепочки инструментов, которые затем будут перечислены в списке выбора цепочек инструментов;
- provide/jpeg.in определяет альтернативные реализации libjpeg;
- provide/openssl.in определяет альтернативные реализации openssl;
- provide/skeleton.in определяет альтернативные реализации скелета;
- provide/init.in определяет альтернативные реализации системы инициализации, это может быть исользовано для выбора скелета по умолчанию для вашиниц ил.

### 9.2.1.5 Контент с свободной формы

Там можно хранить все файлы конфигурации, специфичные для платы, такие как конфигурация ядра, оверлей корневой файловой системы или любой другой файл конфигурации, для которого Buildroot позволяет задать местоположение (используя переменную BR2\_EXTERNAL\_\${NAME}\_PATH). Например, вы можете задать путь к глобальному каталогу патчей, к оверлею rootfs и к файлу конфигурации ядра с ледущим образом (например, запустив make menuconfig и заполнив эти параметры):

```
BR2_GLOBAL_PATCH_DIR=$(BR2_EXTERNAL_BAR_42_PATH)/patches/ BR2_ROOTFS_OVERLAY=$
(BR2_EXTERNAL_BAR_42_PATH)/board/<имя_платы>/overlay/ BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE=$
(BR2_EXTERNAL_BAR_42_PATH)/board/<имя_платы>/kernel.
```

конфигурация

### 9.2.1.6 Дополнительные расширения ядра Linux

Дополнительные расширения ядра Linux (см. раздел 18.22.2) можно добавить, соравив их в каталог linux/ в корне дерева br2-external.

### 9.2.1.7 Пример макета

Вот пример макета, ис пользуую щег о вс е возможнос ти br2-external (пример с одержимог о отображаетс я для файла выше, ког да это уместно) для объяс нения дерева br2-external; это вс е полно тью выдумано только ради иллюстрац ии, конечно):

```
/путь/к/br2-ext-tree/
|- внешний.desc
| | имя BAR_42
| | desc: Пример br2-внешнег о дерева
| |
| |
| | - Конфиг .вх
| |     ис точник "$BR2_EXTERNAL_BAR_42_PATH/toolchain/toolchain-external-mine/Config.in.
| |     параметры"
| |     ис точник "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-1/Config.in"
| |     ис точник "$BR2_EXTERNAL_BAR_42_PATH/package/pkg-2/Config.in"
| |     ис точник "$BR2_EXTERNAL_BAR_42_PATH/package/my-jpeg/Config.in"
| |
| |     конфиг урац ияBAR_42_FLASH_ADDR
| |     | hex "адрес флашпамяти моей платы"
| |     | по умолчанию 0x10AD
| |
| |     ...
| |
| | - внешний.mk
| | include $(sort $(wildcard ${BR2_EXTERNAL_BAR_42_PATH}/package/*/*.mk))
| | include $(sort $(wildcard ${BR2_EXTERNAL_BAR_42_PATH}/toolchain/*/*.mk))

| | flash-my-board:
| |     $(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/flash-image \
| |         --image $(BINARIES_DIR)/image.bin \
| |         --адрес ${BAR_42_FLASH_ADDR}
| |
| |     ...
| |
| | - пакет/pkg-1/Config.in
| | | конфиг урац ияBR2_PACKAGE_PKG_1
| | |     | был "pkg-1"
| | |     | помощь
| | |     | Некоторая помощь по pkg-1
| | |
| | | - пакет/pkg-1/pkg-1.x эш
| | - пакет/pkg-1/pkg-1.mk
| |     |PKG_1_VERSION = 1.2.3
| |     |PKG_1_SITE = /r де/получить/pkg-1
| |     |PKG_1_LICENSE = блабла
| |
| |     |определить PKG_1_INSTALL_INIT_SYSV
| |     | $(INSTALL) -D -m 0755 ${PKG_1_PKGDIR}/S99my-daemon \
| |             | $(TARGET_DIR)/etc/init.d/S99my-daemon
| |     | эндем
| |
| |     |$(eval $(_пакет-autotools))
| |
| |     ...
| |
| |     | - пакет/pkg-1/S99my-daemon
| |
| | - пакет/pkg-2/Config.in
| | - пакет/pkg-2/pkg-2.x эш
| | - пакет/pkg-2/pkg-2.mk
| |
| | - предоставляет/jpeg.in
| |     |config BR2_PACKAGE_MY_JPEG
| |     |     bool "мой jpeg"
| |     |     ...
| | 
```

```
| - пакет/my-jpeg/Config.in
| | конфигурация BR2_PACKAGE_PROVIDES_JPEG
| | по умолчанию "my-jpeg" если BR2_PACKAGE_MY_JPEG
| |
| | - пакет/my-jpeg/my-jpeg.mk
| |    |# Это обычный файл пакета.mk
| |     |MY_JPEG_VERSION = 1.2.3
| |     |MY_JPEG_SITE = https://example.net/some/place
| |     |MY_JPEG_PROVIDES = jpeg
| |     |$(eval $(пакет-autotools))
| |
| |
| | - предоставляет/init.in
| |     |конфигурация BR2_INIT_MINE
| |     |bool "мой пользовательский init"
| |     |выберите BR2_PACKAGE_MY_INIT
| |         выберите BR2_PACKAGE_SKELETON_INIT_MINE, если BR2_ROOTFS_SKELETON_DEFAULT
| |
| |
| | - предоставляет/skeleton.in
| | config BR2_ROOTFS_SKELETON_MINE
| | bool "мой собственный скелет"
| | выберите BR2_PACKAGE_SKELETON_MINE
| |
| | | - package/skeleton-mine/Config.in
| |     |config BR2_PACKAGE_SKELETON_MINE
| |     |бул
| |         выберите BR2_PACKAGE_HAS_SKELTON
| |
| |     |config BR2_PACKAGE_PROVIDES_SKELTON
| |     |по умолчанию "скелет-мина" если BR2_PACKAGE_SKELETON_MINE
| |
| | | - package/skeleton-mine/skeleton-mine.mk
| |     |SKELETON_MINE_ADD_TOOLCHAIN_DEPENDENCY = НЕТ
| |     |SKELETON_MINE_ADD_SKELTON_DEPENDENCY = НЕТ
| |     |SKELETON_MINE_PROVIDES = скелет
| |     |SKELETON_MINE_INSTALL_STAGING = ДА
| |     |$(eval $(generic-package))
| |
| |
| | - provide/toolchains.in
| | config BR2_TOOLCHAIN_EXTERNAL_MINE
| | bool "мой собственный набор инструментов"
| | зависит от BR2_some_arch
| | выберите BR2_INSTALL_LIBSTDCPP
| |
| |
| | | - toolchain/toolchain-external-mine/Config.in.options
| | | если BR2_TOOLCHAIN_EXTERNAL_MINE
| | | config BR2_TOOLCHAIN_EXTERNAL_PREFIX
| | | по умолчанию "arch-mine-linux-gnu"
| | | config BR2_PACKAGE_PROVIDES_TOOLCHAIN_EXTERNAL
| | | конец_если      по умолчанию "toolchain-external-mine"
| |
| |
| | | - toolchain/toolchain-external-mine/toolchain-external-mine.mk
| |     |TTOOLCHAIN_EXTERNAL_MINE_SITE = https://example.net/some/place
| |     |TTOOLCHAIN_EXTERNAL_MINE_SOURCE = мой-инс трументарий.tar.gz
| |     |$(eval $(toolchain-external-package))
| |
| |
| | - linux/Config.ext.in
| | config BR2_LINUX_KERNEL_EXT_EXAMPLE_DRIVER
```

```

||||||-- bool "пример-внешний-драйвер"
помощь
    Пример внешнего драйвера

|- linux/linux-ext-example-driver.mk
|
|- configs/my-board_defconfig
    |BR2_GLOBAL_PATCH_DIR="$(BR2_EXTERNAL_BAR_42_PATH)/патчи"
| |BR2_ROOTFS_OVERLAY="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/overlay/"
| |BR2_ROOTFS_POST_IMAGE_SCRIPT="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/post-
изображение.sh"
| |BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE="$(BR2_EXTERNAL_BAR_42_PATH)/board/my-board/
kernel.config"
|
| |
|- patches/linux/0001-some-change.patch
|- patches/linux/0002-some-other-change.patch
|- патчи/busybox/0001-fix-something.patch
|
|- board/my-board/kernel.config
|- board/my-board/overlay/var/www/index.html
|- доска/модуль/доска/наложение/var/www/my.css
|- доска/модуль/доска/flash-изображение
- board/my-board/post-image.sh
    |#!/бин/ш
    |сгенерировать-мой-бинарный-образ \
    | --root ${BINARIES_DIR}/rootfs.tar \
    | --kernel ${BINARIES_DIR}/zImage \
    | --dtb ${BINARIES_DIR}/my-board.dtb \
    | --output ${BINARIES_DIR}/image.bin
    |
    ...

```

После этого дерево br2-external будет видно в menuconfig (с развернутой компоновкой):

```

Внешние параметры -->
*** Пример дерева br2-external (в /path/to/br2-ext-tree)
[ ] пакет-1
[ ] пакет-2
(0x10AD) адрес флаш-памяти моей платы

```

Если вы используете более одного дерева br2-external, оно будет выглядеть так (с развернутым макетом и вторым с именем FOO\_27, но нет поля desc: в external.desc):

```

Внешние параметры -->
Пример br2-внешнего дерева-->
*** Пример дерева br2-external (в /path/to/br2-ext-tree)
[ ] пакет-1
[ ] пакет-2
(0x10AD) адрес флаш-памяти моей платы
ФУ_27 -->
*** FOO_27 (в /путь/к/другому-br2-ext)
[ ] фу
[ ] бар

```

Кроме того, поставщик JPEG будет виден в выборе JPEG:

```

Целевые пакеты -->
Библиотеки -->
Графика -->
[*] поддержка jpeg
    jpeg вариант () -->
        () jpeg

```

```
( ) jpeg-turbo *** jpeg
      из: Пример br2-external tree ***
(X) мой-jpeg
      *** jpeg из: FOO_27 *** ( ) другой-jpeg
```

И аналогично для наборов инструментов:

```
Цепочка инструментов -->
Набор инструментов () -->
  ( ) Пользовательский набор инструментов
    *** Цепочки инструментов из: Пример br2-external tree ***
(X) мой собственный набор инструментов
```

Примечание. Параметры toolchain в toolchain/toolchain-external-mine/Config.in.options не будут отображаться в меню Toolchain. Они должны быть явно включены из верхнего уровня Config.in br2-external и, таким образом, будут отображаться в меню External options.

## 9.3 Сохранение конфигурации Buildroot

Конфигурация Buildroot можно сохранить с помощью команды make savedefconfig.

Это убирает конфигурацию Buildroot, удаляя параметры конфигурации, которые находятся в своих значениях по умолчанию. Результат сохраняется в файле с именем defconfig. Если вы хотите сохранить другой макет, измените параметр BR2\_DEFCONFIG в самой конфигурации Buildroot или вызовите make с помощью make savedefconfig BR2\_DEFCONFIG=<path-to-defconfig>.

Рекомендуемое место для сохранения defconfig — configs/<boardname>\_defconfig. Если вы последуете этой рекомендации, конфигурация будет указана в make list-defconfigs и ее можно будет снова установить, запустив make <boardname>\_defcon

В качестве альтернативы вы можете скопировать файл в любое другое место и пересобрать его с помощью make defconfig BR2\_DEFCONFIG=<path-to-defcon

## 9.4 Сохранение конфигурации и других компонентов

Файлы конфигурации для BusyBox, ядра Linux, Barebox, U-Boot и uClibc также должны быть сохранены, если они были изменены. Для каждого из этих компонентов существует параметр конфигурации Buildroot, указывающий на их один файл конфигурации, например, BR2\_LINUX\_KERNEL\_CUSTOM. Чтобы сохранить их конфигурацию, задайте эти параметры конфигурации напрямую, поскольку вы хотите отдать сохранение файлов конфигурации, а затем используйте вспомогательные цели, описанные ниже, чтобы фактически сохранить конфигурацию.

Как показано в разделе 9.1, рекомендуемый путь для сохранения этих файлов конфигурации — board/<company>/<boardname>/foo.c

Убедитесь, что вы создали файл конфигурации перед изменением опций BR2\_LINUX\_KERNEL\_CUSTOM\_CONFIG\_FILE и т. д. В противном случае Buildroot попытается получить доступ к этому файлу конфигурации, который еще не существует, и потерпит неудачу. Вы можете создать файл конфигурации, запустив make linux-menuconfig и т. д.

Buildroot предоставляет несколько вспомогательных целей, упрощающих сохранение файлов конфигурации.

- make linux-update-defconfig сохраняет конфигурацию Linux по пути, указанному в BR2\_LINUX\_KERNEL\_CUSTOM\_CONFIG\_FILE. Это упрощает файл конфигурации, удаляя значения по умолчанию. Однако это работает только с ядрами, начиная с 2.6.33. Для более ранних ядер используйте make linux-update-config.
- make busybox-update-config сохраняет конфигурацию busybox по пути, указанному в BR2\_PACKAGE\_BUSYBOX\_CONFIG.
- make uclibc-update-config сохраняет конфигурацию uClibc по пути, указанному в BR2\_UCLIBC\_CONFIG.
- make barebox-update-defconfig сохраняет конфигурацию barebox по пути, указанному в BR2\_TARGET\_BAREBOX\_CUSTOM\_CONFIG.
- make uboot-update-defconfig сохраняет конфигурацию U-Boot по пути, указанному в BR2\_TARGET\_UBOOT\_CUSTOM\_CONFIG.
- Для ядра 91bootstrap3 не существует помощника, поэтому вам придется вручную скопировать файл конфигурации в BR2\_TARGET\_AT91BOOTSTRAP3\_CUSTOM\_CONFIG.

## 9.5 Настстройка сгенерированной целевой файловой системы

Помимо изменения конфигурации с помощью `make *config`, есть еще несколько способов настроить итоговую целевую файловую систему.

Два рекомендуемых метода, которые могут осуществлять, — это наложение(я) корневой файловой системы и скрипты(ы) после сборки.

### Оверлей корневой файловой системы (BR2\_ROOTFS\_OVERLAY)

Наложение файловой системы — это дерево файлов, которое копируется я не посредственно поверх целевой файловой системы после ее создания. Чтобы включить эту функцию, установите параметр конфигурации `BR2_ROOTFS_OVERLAY` (в меню конфигурации системы) в очень наложения.

Вы даже можете указать несколько оверлейов, разделенных пробелом. Если указать относительный путь, он будет относительным к корню дерева Buildroot. Скрытые каталоги и систем контроля версий, такие как .git, .svn, .hg и т. д., файлы с именем .empty и файлы, заканчивающиеся ~, исключаются из копирования.

Если включен `BR2_ROOTFS_MERGED_USR`, то оверлей не должен содержать каталоги /bin, /lib или /sbin, так как Buildroot создаст их как символические ссылки на соответствующие папки в /usr. В такой ситуации, если оверлей имеет какие-либо программы или библиотеки, их следует разместить в /usr/bin, /usr/sbin и /usr/lib.

Как показано в разделе 9.1, рекомендуемый путь для этого оверлея — `board/<company>/<boardname>/rootfs-overlay`.

### Скрипты после сборки (BR2\_ROOTFS\_POST\_BUILD\_SCRIPT)

Скрипты после сборки — это скрипты оболочки, вызываемые после того, как Buildroot собирает все выбранное программное обеспечение, но до сборки образов rootfs.

Чтобы включить эту функцию, укажите разделенный пробелами список скриптов после сборки в параметре конфигурации `BR2_ROOTFS_POST_BUILD` (в меню конфигурации системы). Если указать относительный путь, он будет относительным к корню дерева Buildroot.

Используя скрипты после сборки, вы можете удалить или изменить любой файл в целевой файловой системе. Однако вам следует использовать эту функцию с осторожностью. Всякий раз, когда вы обнаруживаете, что определенный пакет генерирует неправильные или ненужные файлы, вам следует исправить этот пакет, а не обходить его с помощью некоторых скриптов очистки после сборки.

Как показано в разделе 9.1, рекомендуемый путь для этого скрипта — `board/<company>/<boardname>/post_build.sh`.

Скрипты после сборки запускаются с новым деревом Buildroot в качестве текущего рабочего каталога. Путь к целевой файловой системе передается в качестве первого аргумента каждого скрипту. Если параметр конфигурации `BR2_ROOTFS_POST_SCRIPT_ARGS` не пустой, эти аргументы также будут переданы скрипту. Всем скриптам будет передан один и тот же набор аргументов, невозможно передать разные наборы аргументов каждому скрипту.

Обратите внимание, что аргументы из `+BR2_ROOTFS_POST_SCRIPT_ARGS+` также будут переданы в скрипты `post-image` и `post-fakeroot`.

Если вы хотите использовать аргументы, которые используются только для скриптов `post-build`, вы можете использовать `+BR2_ROOTFS_POST_BUILD_SCRIPT_ARGS+`.

+ Кроме того, вы также можете использовать следующие переменные с реди:

- `BR2_CONFIG`: путь к файлу `Buildroot.config`
- `CONFIG_DIR`: каталог, содержащий файл `.config`, и, следовательно, файл `Makefile` Buildroot верхнего уровня для использования (который верен как для сборок в дереве, так и для сборок вне дерева)
- `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`: см. раздел 18.6.2
- `BUILD_DIR`: каталог, в который извлекаются и собираются пакеты
- `BINARIES_DIR`: место, где находятся яркие двоичные файлы (они же изображения)
- `BASE_DIR`: базовый каталог одной директории
- `PARALLEL_JOBS`: количество заданий, используемых при запуске параллельных процессов.

Ниже описаны еще три метода настройки целевой файловой системы, но они не рекомендуются.

### Прямая модификация ядра и файловой системы

Для временных изменений вы можете изменить целевую файловую систему напрямую и перестроить образ. Целевая файловая система дистрибутина в output/target/. После внесения изменений запустите make, чтобы перестроить образ целевой файловой системы.

Этот метод позволяет вам делать чтоугодно с целевой файловой системой, но если вам нужно очистить дерево Buildroot с помощью make clean, эти изменения будут потеряны. Такая очистка необходиима в нескольких случаях, см. раздел [8.2](#) для получения подробной информации. Поэтому это решение полезно только для быстрых тестов: изменения не сохраняются командой make clean. После проверки изменений вы должны убедиться, что они сохраняются после make clean, используя наложение корневой файловой системы или скрипты после сборки.

### Пользовательский целевой скелет (BR2\_ROOTFS\_SKELETON\_CUSTOM)

Образ корневой файловой системы создается из целевого скелета, поверх которого все пакеты устанавливаются со своими файлами. Скелет копируется в целевой каталог output/target до сборки и установки любого пакета. Целевой скелет по умолчанию предоставляет стандартную структуру файловой системы Unix и некоторые базовые сценарии инициализации и конфигурации.

Если скелет по умолчанию (доступный в system/skeleton) не соответствует вашим потребностям, вы обычно используете наложение корневой файловой системы или скрипты после сборки, чтобы адаптировать его. Однако, если скелет по умолчанию полностью отличается от того, что вам нужно, использование пользовательского скелета может быть более подходящим.

Чтобы включить эту функцию, включите опцию конфигурации BR2\_ROOTFS\_SKELETON\_CUSTOM и установите BR2\_ROOTFS\_SKELETON\_CUSTOM на путь к вашему пользователю скелету. Оба параметра доступны в меню конфигурации системы. Если указать относительный путь, он будет относительным к корню дерева Buildroot.

Пользовательские скелеты не обязательно должны содержать каталоги /bin, /lib или /sbin, поскольку они создаются автоматически во время сборки. Если включен BR2\_ROOTFS\_MERGED\_USR, то пользовательский скелет не должен содержать каталоги /bin, /lib или /sbin, поскольку Buildroot создаст их как символические ссылки на соответствующие папки в /usr. В такой ситуации, если скелет имеет какие-либо программы или библиотеки, их следует поместить в /usr/bin, /usr/sbin и /usr/lib.

Этот метод не рекомендуется, поскольку он дублирует весь скелет, что не позволяет воспользоваться яицами правлениями или улучшениями, внесенными в скелет по умолчанию в более поздних выпусках Buildroot.

### Скрипты пост-fakeroot (BR2\_ROOTFS\_POST\_FAKEROOT\_SCRIPT)

При загрузке финальных образов некоторые части процесса требуют прав root: создание узлов устройств в /dev, настройка разрешений или прав собственности на файлы и каталоги и т. д. Чтобы избежать необходимости в реальных правах root, Buildroot использует fakeroot для имитации прав root. Это не полная замена для root, чтобы быть root, но это достаточно для того, что нужно Buildroot.

Скрипты post-fakeroot — это скрипты оболочки, которые вызываются в конце фазы fakeroot, прямо перед вызовом генератора образа файловой системы. Таким образом, они вызываются в контексте fakeroot.

Скрипты Post-fakeroot могут быть полезны в случае, если вам необходимо настроить файловую систему для внесения изменений, которые обычно доступны только пользователю root.

Примечание: рекомендуется использовать существующие механизмы для установки прав доступа к файлам или создания записей в /dev (см. раздел [9.5.1](#)) или для создания пользователей (см. раздел [9.6](#)).

Примечание: различия между скриптами после сборки (выше) и скриптами fakeroot заключаются в том, что скрипты после сборки не вызываются в контексте fakeroot.

Примечание: использование fakeroot не является абсолютной заменой настоящего root. fakeroot подделывает только права доступа к файлам и типы (обычные, блочные или символьные устройства...) и uid/gid; они эмулируются в памяти.

## 9.5.1 Настройка прав доступа к файлам и прав собственности, а также добавление узлов пользовательских устройств

Иногда необходимо задать определенные разрешения или права собственности на файлы или узлы устройств. Например, некоторые файлы могут быть должны принадлежать пользователю root. Поскольку скрипты после сборки не запускаются как root, вы не сможете вносить такие изменения оттуда, если только не используете явный fakeroot из скрипта после сборки.

Вместо этого Buildroot обеспечивает поддержку так называемых таблиц разрешений. Чтобы использовать эту функцию, установите опцию конфигурации BR2\_ROOTFS\_DEVICE\_TABLE и список таблиц разрешений, разделенных пробелами, обычные текстовые файлы, следующие [синтаксису makedev](#).

Если вы используете статическую таблицу устройств (т. е. не используете devtmpfs, mdev или (e)udev), вы можете добавлять узлы устройств, используя тот же синтаксис, в так называемых таблицах устройств. Чтобы использовать эту функцию, установите опцию конфигурации BR2\_ROOTFS\_STATIC\_DEVICE\_TABLE и список таблиц устройств, разделенных пробелами.

Как показано в разделе [9.1](#), рекомендуемое расположение таких файлов — board/<company>/<boardname>/.

Следует отметить, что если определенные разрешения или узлы установлены с вианы с определенным приложением, вместе с этим оно следуем задать переменные FOO\_PERMISSIONS и FOO\_DEVICES в файле .mk пакета (см. раздел 18.6.2).

## 9.6 Добавление пользовательских учетных записей

Иногда необходимо добавить определенных пользователей в целевую систему. Чтобы удовлетворить это требование, Buildroot обеспечивает поддержку так называемых таблиц пользователей. Чтобы использовать эту функцию, установите параметр конфигурации BR2\_ROOTFS\_USERS\_TABLES на список таблиц пользователей, разделенных пробелами, обычные текстовые файлы, следующие синтаксису makeusers.

Как показано в разделе 9.1, рекомендуемое расположение таких файлов — board/<company>/<boardname>/.

Следует отметить, что если пользовательские пользователи с вианы с определенным приложением, вместе с этим оно следуем установить переменную FOO\_USERS в файле .mk пакета (см. раздел 18.6.2).

## 9.7 Настойка после создания изображений

В то время как скрипты пост-баки (раздел 9.5) запускаются перед бакой образа файловой системы, ядра и загрузчика, скрипты после баки образа могут использоваться для выполнения некоторых определенных действий после создания всех образов.

Например, скрипты пост-образа можно использовать для автоматического извлечения архива корневой файловой системы в место расположение, экспортируя вашим NFS-сервером, или для создания пакетного образа прошивки, объединяя его корневую файловую систему и образ ядра, или для любог другого пользователя для действий необязательно для вашего проекта.

Чтобы включить эту функцию, укажите разделенный пробелами список скриптов пост-образа в параметре конфигурации BR2\_ROOTFS\_POST\_IMAGE\_SCRIPT (в меню конфигурации). Если указывать относительный путь, он будет относительным к корню дерева Buildroot.

Как и скрипты post-build, скрипты post-image запускаются с новым деревом Buildroot в качестве текущего рабочего каталога. Путь к вых одному каталогу изображений передается в качестве первого аргумента каждому скрипту. Если параметр конфигурации BR2\_ROOTFS\_POST\_SCRIPT\_ARGS не пустой, эти аргументы также будут переданы скрипту. Всем скриптам будет передан один и тот же набор аргументов, невозможно передать разные наборы аргументов каждому скрипту.

Обратите внимание, что аргументы из BR2\_ROOTFS\_POST\_SCRIPT\_ARGS также будут переданы в скрипты post-build и post-fakeroot.

Если вы хотите использовать аргументы, которые используются только для скриптов пост-изображения, вы можете использовать BR2\_ROOTFS\_POST\_IMAGE\_SCRIPT\_ARGS.

Однако, как и в случае с скриптами пост-баки, скрипты имеют доступ к переменным среды BR2\_CONFIG, HOST\_DIR, STAGING\_DIR, TARGET\_DIR, BUILD\_DIR, BINARIES\_DIR, CONFIG\_DIR, BASE\_DIR и PARALLEL\_JOBS.

Скрипты post-image будут выполняться ятом имени пользователя, который запускает Buildroot, что обычно не должно быть пользователем root. Поэтому любое действие, требующее прав root в одном из этих скриптов, потребует специальной обработки (использование fakeroott или sudo), которая оставляет науш мониторинг разработчика скрипта.

## 9.8 Добавление патчей и эшей для конкретного проекта

### 9.8.1 Предоставление дополнительных исправлений

Иногда полезно применять дополнительные патчи к пакетам - поверх тех, что предоставлены в Buildroot. Это может быть использовано для поддержки пользовательских функций в проекте, например, или при работе над новой версией ядра.

Параметр конфигурации BR2\_GLOBAL\_PATCH\_DIR можно использовать для указания списка из одного или нескольких каталогов, разделенных пробелами, содержащих исправления пакетов.

Для определенной версии <packageversion> определенного пакета <packagename> исправления применяются из BR2\_GLOBAL\_PATCH с помощью образом:

1. Для каждого каталога - <global-patch-dir> - который существует в BR2\_GLOBAL\_PATCH\_DIR, <package-patch-dir> будет определяться ясльшим образом:

- <global-patch-dir>/<packagename>/<packageversion>/, если каталог существует. • В противном случае, <global-patch-dir>/<packagename>, если каталог существует.

2. Затем ис правления будут применяться из <package-patch-dir> следующим образом:

- Если файл сери существуют в каталоге пакета, то патчи применяются к соответствию с файлом серии; • В противном случае файлы патчей, соответствующие \*.patch, применяются в алфавитном порядке. Поэтому, чтобы гарантировать их применение в правильном порядке, настоятельно рекомендуется называть файлы патчей следующим образом: <номер>-<описание>.patch, где <номер> относится к порядку применения.

Информацию о том, как применяться ис правления для пакета, см. в разделе 19.2.

Параметр BR2\_GLOBAL\_PATCH\_DIR является я предпочтительным методом указания пользовательского каталога патчей для пакетов. Его можно использовать для указания каталога патчей для любого пакета в buildroot. Его также следует использовать вместе с параметром пользовательского каталога патчей, которые доступны для таких пакетов, как U-Boot и Barebox. Благодаря этому пользователь сможет управлять своими патчами из одного каталога верхнего уровня.

Использованием из BR2\_GLOBAL\_PATCH\_DIR, который является я предпочтительным методом для указания пользовательских патчей, является я BR2\_LINUX\_KERNEL\_P BR2\_LINUX\_KERNEL\_PATCH с следует использовать для указания патчей ядра, которые доступны по URL. Примечание: BR2\_LINUX\_KERNEL\_ указывает патчи ядра, которые применяются я после патчей, доступных в BR2\_GLOBAL\_PATCH\_DIR, как это делается из ука post-patch пакета Linux.

#### 9.8.2 Предоставление дополнительных хешей

Buildroot объединяет с списком хешей, по которым он проверяет целостность загруженных архивов или тех, которые он генерирует локально из VCS checkouts. Однако он может делать это только для известных версий; для пакетов, где можно указать пользовательскую версию (например, пользовательскую строку версии, URL-адрес удаленного tarball или местоположение репозитория VCS и набор изменений), Buildroot не может перенести хеш для них.

Для пользователей, обесценивших целостностью таких загрузок, можно предоставить список хешей, которые Buildroot может использовать для проверки произвольных загруженных файлов. Эти дополнительные хеши ищутся аналогично дополнительным патчам (выше); для каждого каталога в BR2\_GLOBAL\_PATCH\_DIR первый существующий файл используется для проверки загрузки пакета.

- <global-patch-dir>/<packagename>/<packageversion>/<packagename>.hash
- <global-patch-dir>/<packagename>/<packagename>.hash

Для генерации этих файлов можно использовать скрипты utils/add-custom-hashes.

#### 9.9 Добавление пакетов, сецифичных для проекта

Как правило, любой новый пакет следует добавлять непосредственно в каталог пакетов и отправлять в вышеупомянутый проект Buildroot.

Как добавлять пакеты в Buildroot в целом, подробно объясняется в главе 18 и не будет повторяться здесь. Однако, вашему проекту могут понадобиться некоторые фирменные пакеты, которые нельзя передать в upstream. В этом разделе будет объяснено, как можно хранить такие пакеты, сецифичные для проекта, в каталоге, сецифичном для проекта.

Как показано в разделе 9.1, рекомендуемое местоположение для пакетов, сецифичных для проекта, — package/<company>/. Если вы используете функцию дерева br2-external (см. раздел 9.2), рекомендуемое местоположение — поместить их в подкаталог с именем package/ в вашем дереве br2-external.

Однако Buildroot не будет знать о пакетах в этом месте, если мы не выполним некоторые дополнительные шаги. Как объяснялось в главе 18, пакет в Buildroot в основном состоит из двух файлов: файла.mk (описывающего, как собрать пакет) и файла Config.in (описывающего параметры конфигурации для этого пакета).

Buildroot автоматически включает файлы .mk в подкаталог первого уровня каталога пакета (используя шаблон package/\*/\*.mk). Если мы хотим, чтобы Buildroot включил файлы .mk из более глубоких подкаталогов (например, package/<company>/package1), то нам просто нужно добавить файл .mk в подкаталог первого уровня, который включает эти дополнительные файлы .mk. Поэтому создайте файл package/<company>/<company>.mk с следующим содержимым (предполагая, что у вас есть только один дополнительный уровень каталога ниже package/<company>/):

```
включить $(sort $(wildcard package/<company>/*/*.mk))
```

Для файлов Config.in с создайте файл package/<company>/Config.in, который включает файлы Config.in всех ваших пакетов. Необходимо предстатьвить исчерпывающий список ок, поскольку подстановочные знаки не поддерживаются в команде source kconfig. Например:

```
ис точник "package/<компания>/package1/Config.in" ис точник  
"package/<компания>/package2/Config.in"
```

Включите этот новый файл package/<company>/Config.in из package/Config.in, желательно в меню для конкретной компании, чтобы упростить слияние с будущими версиями Buildroot.

Если вы используете дерево br2-external, обратитесь к разделу [9.2](#), чтобы узнать, как заполнить эти файлы.

## 9.10 Краткое руководство по созданию настроек, специфичных для вашего проекта

Ранее в этой главе были описаны различные методы создания настроек, специфичных для проекта. В этом разделе мы обобщим все это, предстартив пошаговые инструкции по созданию настроек, специфичных для проекта. Очевидно, что шаги, которые не имеют отношения к вашему проекту, можно пропустить.

1. Создайте menuconfig для настроек ядра и инструментов, пакетов и ядра.
2. Создайте linux-menuconfig для обновления конфигурации ядра, аналогично для других конфигураций, таких как busybox, uclibc, ...
3. mkdir -p board/<производитель>/<имя платы>
4. Установите следующие параметры в board/<производитель>/<имя\_платы>/<пакет>.config (если они соответствуют):

- BR2\_LINUX\_KERNEL\_CUSTOM\_CONFIG\_FILE
- BR2\_PACKAGE\_BUSYBOX\_CONFIG
- BR2\_UCLIBC\_CONFIG
- BR2\_TARGET\_AT91BOOTSTRAP3\_CUSTOM\_CONFIG\_FILE
- BR2\_TARGET\_BAREBOX\_CUSTOM\_CONFIG\_FILE
- BR2\_TARGET\_UBOOT\_CUSTOM\_CONFIG\_FILE

5. Напишите файлы конфигурации:

- сделать linux-update-defconfig
- сделать busybox-update-config
- сделать uclibc-update-config
- cp <все данные>/build/at91bootstrap3-\*.config board/<производитель>/<имя\_платы>/at91boot
- сделать barebox-update-defconfig
- сделать uboot-update-defconfig

6. Создайте board/<manufacturer>/<boardname>/rootfs-overlay/ и заполните его дополнительными файлами, которые вам нужны. Ваш rootfs, например board/<manufacturer>/<boardname>/rootfs-overlay/etc/inittab. Установите BR2\_ROOTFS\_O в board/<manufacturer>/<boardname>/rootfs-overlay.

7. Создайте скрипт пост-борд board/<manufacturer>/<boardname>/post\_build.sh. Установите BR2\_ROOTFS\_POST\_BUILD в board/<производитель>/<название\_платы>/post\_build.sh

8. Если необходимо установить дополнительные разрешения setuid или создать узлы устройств, создайте board/<manufacturer>/<boardname> и добавьте этот путь в BR2\_ROOTFS\_DEVICE\_TABLE.

9. Если необходимо создать дополнительные учетные записи пользователей, создайте `board/<manufacturer>/<boardname>/users_table.txt` и добавьте этот путь в `BR2_ROOTFS_USERS_TABLES`.
10. Чтобы добавить пользовательские патчи к определенным пакетам, установите `BR2_GLOBAL_PATCH_DIR` в `board/<manufacturer>/<boardname>/` и добавьте с вашими патчами для каждого пакета в подкаталог с именем пакета. Каждый патч должен называться я`<packagename>`
11. Специально для ядра Linux также существует опция `BR2_LINUX_KERNEL_PATCH`, с новым префиксом в котором отображается возможность загрузки патчей с URL. Если вам это не нужно, предпочтительнее использовать `BR2_GLOBAL_PATCH_DIR`. U-Boot, Barebox, at91bootstrap и at91bootstrap3 также имеют отдельные опции, но они не дают никаких преимуществ перед `BR2_GLOBAL_PATCH_DIR` и, скорее всего, будут удалены в будущем.
12. Если вам нужно добавить пакеты, специфичные для проекта, создайте `package/<manufacturer>/` и поместите ваши пакеты в этот каталог. Создайте общий файл `<manufacturer>.mk`, который включает файлы `.mk` всех ваших пакетов. Создайте общий файл `Config.in`, который является источником файлов `Config.in` всех ваших пакетов. Включите этот файл `Config.in` из файла `package/Config.in` в `Buildroot`.
13. сделайте `savedefconfig`, чтобы сохранить конфигурацию `buildroot`.
14. `cp defconfig configs/<имя_платы>_defconfig`

# Глава 10

## Интеграционные темы

В этой главе обсуждаются различные вещи интегрируются на системном уровне. Buildroot обладает высокой степенью настройки, почти все, что здесь обсуждается, может быть изменено или переопределено с помощью [наложения rootfs](#) или [пользовательской конфигурации](#) келета.

### 10.1 Настраиваемые пакеты

Некоторые базовые пакеты, такие как Busybox и uClibc, можно настраивать с определенными функциями или без них. При написании кода Buildroot, ис пользуя щег отакие пакеты, участник может предполагать, что опции, включенные в конфигурациях, предоставляемых Buildroot, включены. Например, package/busybox/busybox.config устанавливает CONFIG\_FEATURE\_START\_STOP\_DAEMON\_LONG\_OPTIO, поэтому сценарии инициализации, предназначенные для использования Busybox init, могут использовать start-stop-daemon с длинными параметрами.

Люди, ис пользуя щие пользовательские конфигурации, которые отключают такие параметры по умолчанию, несут ответственность за то, чтобы все соответствующие скрипты/пакеты продолжали работать, если нет, то за их соответствующую адаптацию. Чтобы следовать примеру Busybox выше, отключение параметров длинной формы потребует замены скриптов инициализации, которые их ис пользуют (в наложении).

### 10.2 Системный

В этой главе описываются решения принятые при интеграции systemd в Buildroot, а также способы реализации различных вариантов ис пользования.

#### 10.2.1 Демон DBus

Systemd требует демона DBus. Для него есть два варианта: традиционный dbus (BR2\_PACKAGE\_DBUS) и bus1 dbus-broker (BR2\_PACKAGE\_DBUS\_BROKER). Необходимо выбрать хотя бы один из них. Если оба включены в конфигурацию, dbus-broker будет ис пользоваться как системная машина, но традиционный dbus-daemon все равно будет установлен и может ис пользоваться как сервис машины. Также можно ис пользовать его инструменты (например, dbus-send) (если systemd имеет busctl в качестве альтернативы). Кроме того, традиционный пакет dbus — единственный, который предоставляет libdbus, который ис пользуется многими пакетами как библиотеками интеграции dbus.

И в случае dbus, и в случае dbus-broker демон работает как пользователь dbus. Файлы конфигурации DBus также идентичны для обоих.

Чтобы гарантировать, что только один из двух демонов запускается как система машина, файлы активации systemd пакета dbus (dbus.socket и символический ссылка dbus.service в multi-user.target.wants) удаляются при выборе dbus-broker.

### 10.3 Использование SELinux в Buildroot

SELinux это модуль безопасности ядра Linux, обеспечивающий соблюдение политики контроля доступа. В дополнение к традиционным разрешениям файлов и символьическим ссылкам доступа SELinux позволяет писать правила для пользователей или процессов для доступа к определенным функциям ресурсов (файлов, скриптов...).

SELinux имеет три режима работы:

- Отключено: политика не применяется
- Разрешительный: политика применяется к несанкционированные действиям построения. Этот режим часто используется для устранения неполадок с SELinux.
- Принудительное применение: политика применяется и несанкционированные действия запрещены.

В Buildroot режим работы контролируется параметрами конфигурации BR2\_PACKAGE\_REFPOLICY\_POLICY\_STATE\_\*. Ядро Linux также имеет различные параметры конфигурации, которые влияют на включение SELinux (см. security/selinux/Kconfig в исходниках ядра Linux).

По умолчанию в Buildroot политика SELinux предоставляется вышеупомянутой [refpolicy](#), проект, включенный с помощью BR2\_PACKAGE\_REFPOLICY.

### 10.3.1 Включение поддержки SELinux

Для обеспечения надлежащей поддержки SELinux в системе, созданной с помощью Buildroot, необходимо включить следующие параметры конфигурации:

- BR2\_PACKAGE\_LIBSELINUX
- BR2\_PACKAGE\_REFPOLICY

Кроме того, формат образа вашей файловой системы должен поддерживать расширенные атрибуты.

### 10.3.2 Тонкая настройка политики SELinux

SELinux refpolicy содержит модули, которые можно включать и отключать при сборке. Каждый модуль предоставляет ряд правил SELinux. В Buildroot небазовые модули отключены по умолчанию, и предустановлены несколько способов включения таких модулей:

- Пакеты могут включать список модулей SELinux в refpolicy с помощью <packagename>\_SELINUX\_MODULES переменная.
- Пакеты могут предавлять дополнительные модули SELinux, помещая их (файлы .fc, .if и .te) в package/<packagename>/selinux.
- Дополнительные модули SELinux можно добавлять в каталог, указанный в конфигурации BR2\_REFPOLICY\_EXTRA\_MODULES\_DIRS. вариант пайка.
- Дополнительные модули в refpolicy могут быть включены, если они указаны в BR2\_REFPOLICY\_EXTRA\_MODULES\_DEPENDENCIES возможность настройки.

Buildroot также позволяет полностью переопределить refpolicy. Это позволяет предавать полную настраиваемую политику, разработанную специально для данной системы. При использовании этого способа вышеупомянутые механизмы отключаются в политике, не добавляется ни один дополнительный модуль SELinux, а все доступные модули в настраиваемой политике включаются в финальную бинарную политику. Настраиваемая политика должна быть ответвлением официальной [refpolicy](#).

Чтобы полностью переопределить refpolicy, необходимо задать следующие переменные конфигурации:

- BR2\_PACKAGE\_REFPOLICY\_CUSTOM\_GIT
- BR2\_PACKAGE\_REFPOLICY\_CUSTOM\_REPO\_URL
- BR2\_PACKAGE\_REFPOLICY\_CUSTOM\_REPO\_VERSION

# Глава 11

## Часто задаваемые вопросы и устранение неполадок

### 11.1 Загрузка зависает после запуска сети . . .

Если процесс загрузки зависает после следующих сообщений (сообщения не обязательно полностью совпадают, в зависимости от списка выбранных пакетов):

Освобождение памяти инициализации:

3972 КБ Инициализация ядра инициатора случайных чисел... выполнено.

Запуск сети...

Запуск dropbear sshd: генерация ключа rsa... генерация ключа dsa... OK

то это означает, что ваша система работает, но не запустила оболочку на последовательной консоли. Чтобы система запустила оболочку на вашей последовательной консоли, вам нужно перейти в конфигурацию Buildroot, в конфигурации системы изменить Run a getty (login prompt) after boot и установить соответствующий порт и способ передачи в подменю параметров getty. Это автоматически настраивает файл /etc/inittab сгенерированной системы так, чтобы оболочка запускалась на правильном последовательном порту.

### 11.2 Почему на целевой платформе нет компилятора?

Было решено, что поддержка собственного компилятора на целевой платформе будет прекращена с версией Buildroot-2012.11 по следующим причинам:

- эта функция не поддерживалась, не тестировалась и часто не работала;
- эта функция была доступна только для наборов инструментов Buildroot;
- Buildroot в основном нацелен на небольшое или очень небольшое целевое оборудование с ограниченными ресурсами на борту (ЦПОУ, запоминающее устройство), для которых компиляция на целевом устройстве не имеет особого смысла;
- Buildroot направлен на проектирование кросскомпиляции, делая не нужной собственную компиляцию на целевой платформе.

Если вам в любом случае нужен компилятор на вашей цели, то Buildroot не подходит для ваших целей. В таком случае вам нужен настоящий дистрибутив, и вам следует выбрать что-то вроде:

- открытое встраиваемое
- Йокто
- Дебиан
- Федора
- openSUSE ARM
- Арч Линукс ARM
- ...

## 11.3 Почему на целевой системе нет файлов разработки?

Поскольку на целевой платформе нет доступного компилятора (см. раздел 11.2), нет способа отладить местонахождение файлов или статические библиотеки.

Поэтому эти файлы всегда даются из ядра с момента выпуска Buildroot-2012.11.

## 11.4 Почему на цель нет документации?

Поскольку Buildroot в основном ориентирован на небольшое или очень небольшое целевое оборудование с ограниченными выделенными ресурсами (ЦП, оперативная память, запоминающее устройство), нет способа отладить местонахождение документации.

Если вам в любом случае нужны данные документации по вашей цели, то Buildroot для вас не подходит, и вам следует поискать настоящий дистрибутив (см.: Раздел 11.2).

## 11.5 Почему некоторые пакеты не видны в меню конфигуратора Buildroot?

Если пакет существует в дереве Buildroot и не отображается в меню конфигуратора, это, скорее всего, означает, что некоторые зависимости пакета не выполнены.

Чтобы узнать больше о зависимостях пакета, найдите символ пакета в меню конфигуратора (см. раздел 8.1).

Затем вам, возможно, придется рекурсивно включить несколько опций (с соответствующими неудовлетворенными зависимостями), чтобы наконец получить возможность выбрать пакет.

Если пакет не виден из-за некоторых неудовлетворенных параметров целочки инструментов, то вам обязательно следует выполнить полную переборку (более подробные объяснения см. в разделе 8.1).

## 11.6 Почему бы не использовать целевой каталог в качестве chroot-каталога?

Существует множество причин не использовать целевой каталог как chroot-каталог, среди которых:

- в целевом каталоге неправильно установлены права собственности на файлы, режимы и разрешения
- узлы устройства не создаются в целевом каталоге.

По этим причинам команды, запущенные через chroot, использующие целевой каталог в качестве нового корня с корнем всеего, не будут работать.

Если вы хотите запустить целевую файловую систему внутри chroot или как корень NFS, то используйте образ tarball, созданный в images/, и извлеките его как корневой каталог.

## 11.7 Почему Buildroot не генерирует двоичные пакеты (.deb, .ipkg...)?

Одна из функций, которая часто обсуждается в списке Buildroot, — это общая тема «управления пакетами». Подводя итог, можно сказать, что идея заключается в том, чтобы добавить некоторое отслеживание того, какой пакет Buildroot устанавливает какие файлы, с целями:

- возможность удаления файлов, установленных пакетом, когда этот пакет отменяется в menuconfig;
- возможность генерировать двоичные пакеты (ipk или другой формат), которые могут быть установлены на целевой системе без повторной генерации нового корня образа файловой системы.

В целом, большинство людей думают, что это легкое дело: просто отследить, какой пакет что установил, и удалить это, когда пакет не выбран. Однако, все гораздо сложнее:

- Речь идет не только о каталоге target/, но и sysroot в host/<tuple>/sysroot и самом каталоге host/. Всё файлы, установленные в этих каталогах различными пакетами, должны отслеживаться.
- Когда пакет отменяет выбор из конфигурации, недостаточно удалить только файлы, которые он установил. Необходимо также удалить все его обратные зависимости (т. е. пакеты, полагающиеся на него) и пересобрать все эти пакеты. Например, пакет A зависит необязательно от библиотеки OpenSSL. Выбирают яоба, и создается Buildroot. Пакет A создается яс поддержкой шифрования с использованием OpenSSL. Позже OpenSSL отменяется из конфигурации, но пакет A остается (поскольку OpenSSL является явно обязательной зависимостью, это возможно). Если удаляются только файлы OpenSSL, то файлы, установленные пакетом A, оказываются сломанными: они используют библиотеку, которая больше не присутствует на целевом объекте. Хотя это технически выполнимо, это значительно усложняет Buildroot, что противоречит цели, которой мы стараемся придерживаться.
- В дополнение к предыдущей проблеме, есть случай, когда не обязательная зависимость даже не известна Buildroot. Например, пакет A в версии 1.0 никогда не использовал OpenSSL, но в версии 2.0 он автоматически использует OpenSSL, если он доступен. Если файл Buildroot.mk не был обновлен, чтобы учесть это, то пакет A не будет частью обратных зависимостей OpenSSL и не будет удален и перестроен при удалении OpenSSL. Конечно, файл .mk пакета A должен быть исправлен, чтобы упомянуть эту необязательную зависимость, но в тоже время вы можете иметь неожиданное поведение.
- Запрос также заключается в том, чтобы разрешить применение изменений в menuconfig к вых одному каталогу без необходимости перестраивать всё с нуля. Однако это очень трудно добиться надежным способом: что происходит при изменении подопечного пакета (нам придется бы обнаружить это и перестроить пакет с нуля, возможно, все его обратные зависимости), что происходит при изменении опций toolchain и т. д. На данный момент то, что делает Buildroot, ясно и просто, поэтому его поведение очень надежно и олегчает поддержку пользователям. Если изменения в конфигурации, внесенные в menuconfig, применяются после следующего make, то он должен работать правильно и надлежащим образом во всех случаях, а не иметь каких-либо странностей в крайних случаях. Риск заключается в получении отчетов об ошибках типа «Я включил пакеты A, B и C, затем запустил make, затем отключил пакет C и включил пакет D и запустил make, затем снова включил пакет C и включил пакет E, а затем происходит сбой с борком». Или еще хуже: «Я сделал некоторую конфигурацию, затем выполнил с борком, затем внес некоторые изменения, выполнил с борком, еще некоторые изменения выполнил с борком, и теперь он падает, но я не помню все изменения, которые я делал, и в каком порядке». Это будет невозможно поддерживать.

После этого причин можно сделать вывод, что добавление отслеживания установленных файлов для их удаления при отмене выбора пакета или для создания репозитория двойных пакетов — это то, чего очень трудно добиться и это значительно усложняет задачу.

Поэтому по этому поводу разработчики Buildroot делают следующее заявление:

- Buildroot стремится упростить создание корневой файловой системы (откуда и название, кстати). Это то, что мы хотим сделать. Buildroot оправдан: построение корневых файловых систем.
- Buildroot не предназначен для того, чтобы быть дистрибутивом (или, с корнем, генератором дистрибутивов). Большинство разработчиков Buildroot считают, что это не цель, к которой мы должны стремиться. Мы считаем, что есть другие инструменты, которые лучше подходят для создания дистрибутива, чем Buildroot. Например, [Open Embedded](#), или [openWRT](#), являются такими инструментами.
- Мы предпочитаем продвигать Buildroot в направлении, которое упрощает (или даже упрощает) создание полных корневых файловых систем. Это то, что выделяет Buildroot из толпы (помимо прочего, конечно!).
- Мы считаем, что для большинства систем Linux бинарные пакеты не нужны и потенциально вредны. Использование бинарных пакетов означает, что система может быть частично обновлена, что создает огромное количество возможных комбинаций версий пакетов, которые следуют протестированием перед выполнением обновления настраиваемой структуре. С другой стороны, выполняя полное обновление системы путем обновления всех его образа корневой файловой системы с разом, образ, развернутый на настраиваемой системе, гарантированно будет действительно протестированным и проверенным.

## 11.8 Как ускорить процесс с борками?

Поскольку Buildroot часто подразумевает выполнение полной перестройки всей системы, что может занять довольно много времени, ниже мы приводим ряд советов, которые помогут сократить время с борками:

- Используйте готовый внешний набор инструментов вместе с внутренним набором инструментов Buildroot по умолчанию. Используйте готовый набор инструментов Linaro (на ARM) или набор инструментов Sourcery CodeBench (для ARM, x86, x86-64, MIPS и т. д.), вы сэкономите время с борками набора инструментов при каждой полной перестройке, примерно 15–20 минут. Обратите внимание, что временное использование внешнего набора инструментов не мешает вам переключаться обратно на внутренний набор инструментов (который может обеспечить более высокий уровень настройки), как только осталась часть вашей системы заработает;

- Используйте кеш компилятора `ccache` (см. раздел [8.13.3](#));
- Узнайте о пересборке только тех нескольких пакетов, которые вам действительно нужны (см. Раздел [8.3](#)), но помните, что иногда в любом случае необходиима полная пересборка (см. Раздел [8.2](#));
- Убедитесь, что вы не используете виртуальную машину для системы Linux, ис пользуемой для запуска Buildroot. Большинство этих нологий виртуальных машин, как известно, оказывают значительное влияние на производительность ввода-вывода, что действительно важно для сборки и их одногокод;
- Убедитесь, что вы используете только локальные файлы: не пытайтесь выполнить сборку через NFS, это значительно замедлит сборку. Наличие локальной папки загрузки Buildroot также немногом помогает.
- Купите новое оборудование. SSD и большой объем оперативной памяти — ключ к ускорению сборки.
- Пожалуйста, экспериментируйте с параллельной сборкой верхнего уровня с см. раздел [8.12](#).

## 11.9 Как Buildroot поддерживает Y2038?

Следует рассмотреть несколько ситуаций:

- На 64-битных архитектурах проблем не возникает, поскольку `time_t` всегда был 64-битным.
- На 32-битных архитектурах ситуация зависит от библиотеки C:
  - С `uclibc-ng` есть поддержка 64-битного `time_t` на 32-битных архитектурах, начиная с версии 1.0.46, поэтому системы, использующие `uclibc-ng` на 32-битных платформах, будут совместимы с Y2038, когда `UCLIBC_USE_TIME64` равен 1. Это значение по умолчанию с версией 1.0.49.
  - С `musl` 64-битный `time_t` всегда использовался на 32-битных архитектурах, поэтому системы, использующие `musl` на 32-битных платформах, совместимы с Y2038.
  - С `glibc` 64-битный `time_t` на 32-битных архитектурах включается языком `Buildroot BR2_TIME_BITS_64`. При включении этой опции в системе, использующей `glibc` на 32-битных платформах, будет совместима с Y2038.

Обратите внимание, что выше приведены только комментарии о возможностях библиотеки C. Отдельные библиотеки или приложения пользователя могут просматривать даже если они созданы в конфигурации, совместимой с Y2038, могут демонстрировать некорректное поведение, если они неправильно используют API и типы времени.

# Глава 12

## Известные проблемы

- Не возможно передать дополнительные параметры компоновщика через BR2\_TARGET\_LDFLAGS, если такие параметры содержат знак \$. Например, известно, что следующее приводит к сбою: BR2\_TARGET\_LDFLAGS="-Wl,-rpath=\"\$ORIGIN/../lib\""
- Пакет libffi не поддерживается на архитектурах SuperH 2 и ARMv7-M.
- Пакет prboom вызывает сбой компиляции с компилятором SuperH 4 из Sourcery CodeBench версии 2012.09.

# Глава 13

## Юридическое уведомление и лицензирование

### 13.1 Соблюдение лицензий с открытым исходным кодом

Все конечные продукты Buildroot (инструментарий, корневая файловая система, ядро, загрузчики) содержат программное обеспечение с открытым исходным кодом, выпущенное под различными лицензиями.

Использование программного обеспечения с открытым исходным кодом дает вам свободу сдавать мощные системы, выбирая из широкого спектра пакетов, но также налагает некоторые обязательства, которые вы должны знать и соблюдать. Некоторые лицензии требуют, чтобы вы опубликовали текст лицензии в документации вашего продукта. Другие требуют, чтобы вы распространяли исходный код программного обеспечения, кто получает ваш продукт.

Точные требования каждой лицензии задокументированы в каждом пакете, и вы (или ваш юридический офис) несете ответственность за соблюдение этих требований. Чтобы облегчить вам задачу, Buildroot может обратиться к вам некоторые материалы, которые вам, вероятно, понадобятся. Чтобы создать эти материалы, после того, как вы настроили Buildroot с помощью make menuconfig, make xconfig или make gconfig, выполните:

сделать юридическую информацию

Buildroot собирает юридически значимый материал в ваших одном каталоге в подкаталоге legal-info/. Там вы найдете:

- Файл README, который описывает созданный материал и содержит предупреждения о материалах, которые Buildroot не смог создать.
- buildroot.config: это файл конфигурации Buildroot, который обычно создается с помощью make menuconfig и который необходимо воспроизвести с борком.

Исходный код для всех пакетов; он сохраняется в подкаталогах sources/ и host-sources/ для целевых и хостовых пакетов с соответственно. Исходный код для пакетов, которые устанавливают <PKG>\_REDISTRIBUTE = NO, не будет сохранен. Также сохраняются примененные исправления вместе с файлом с именем series, в котором перечислены исправления в порядке их применения.

Патчи находятся под той же лицензией, что и файлы, которые они изменяют. Примечание: Buildroot применяет дополнительные патчи к скрипту Libtool пакетов на основе autotools. Эти патчи можно найти в support/libtool в исходном коде Buildroot, и из-за технических различий они не сохраняются вместе с исходными кодами пакетов. Возможно, вам придется скопировать их вручную.

- Файл манифеста (один для целевых и один для хостовых пакетов), в котором перечислены настроенные пакеты, их версии, лицензии и связанные с ними информации. Часть этой информации может быть не определена в Buildroot; такие элементы помечены как «неизвестные».
- Тексты лицензий всех пакетов в подкаталогах licenses/ и host-licenses/ для целевых и хостовых пакетов с соответственно. Если файл(ы) лицензии не определены в Buildroot, файл не создается, и предупреждение в README указывает на это.

Обратите внимание, что есть функции юридической информации Buildroot — предстаивать все материалы, которые так или иначе имеют отношение к юридическому соблюдению лицензий пакета. Buildroot не пытается представить точный материал, который вы должны каким-то образом сделать

public. Конечно, производится я больше материала, чем требуется для ядра с соблюдением закона. Например, он производит исходный код для пакетов, выпущенных под лицензиями типа BSD, которые вы не обязаны распространять в исходном виде.

Более того, из-за технических ограничений Buildroot не создает некоторые материалы, которые вам могут понадобиться например, исходный код toolchain для некоторых внешних toolchain и с исходным кодом Buildroot. Когда вы запускаете make legal-info, Buildroot создает предупреждения в файле README, чтобы сообщить вам о соответствии юридического материала, который не удалось сохранить раннее.

Наконец, имейте в виду, что вывод make legal-info основан на декларативных утверждениях в каждом из рецептов пакетов.

Разработчики Buildroot стараются сделать все возможное, чтобы эти декларативные заявления были максимально точными, насколько это возможно. Однако вполне возможно, что эти декларативные заявления не полностью точны и не являются ясно читаемыми. Вы (или ваш юридический отдел) должны проверить вывод make legal-info, прежде чем использовать его в качестве собственной документации См. пункты NO WARRANTY (пункты 11 и 12) в файле COPYING в корне дистрибутива Buildroot.

## 13.2 Соблюдение лицензии Buildroot

Сам Buildroot — это программное обеспечение с открытым исходным кодом, выпущенное по [лицензии GNU General Public License версии 2](#), или (по вашему выбору) любая более поздняя версия, за исключением правил пакетов, подробно описаных ниже. Однако, будучи системой сборки, она обычно не является частью конечного продукта, если вы разрабатываете корневую файловую систему, ядро, загрузчик или набор инструментов для устройства, код Buildroot присутствует только на машине разработки, а не в релизах устройств.

Тем не менее, общее мнение разработчиков Buildroot заключается в том, что при выпуске продукта, содержащего программное обеспечение под лицензией GPL, следует выпускать исходный код Buildroot вместе с исходным кодом других пакетов. Это связано с тем, что [GNU GPL](#) определяет «полный исходный код» для исключения работы как «весь исходный код для всех модулей», которые он содержит, плюс любые связанные файлы определения интерфейса, плюс скрипты, используемые для управления компиляцией и установкой исключениями файла. Buildroot является частью скриптов, используемых для управления компиляцией и установкой исключениями файла, и как таковой он считается частью материала, который должен быть распространен.

Помните, что это всего лишь мнение разработчиков Buildroot, и в случае возникновения каких-либо сомнений вам следует проконсультироваться с вашим юридическим отделом или юристом.

### 13.2.1 Патчи для пакетов

Buildroot также объединяет файлы исключений, которые применяются к исходникам различных пакетов. Эти исключения не покрывают ядра лицензией Buildroot. Вместо этого они покрывают ядра лицензией программного обеспечения к которому применяются исключения. Когда указанное программное обеспечение доступно по нескольким лицензиям, исключения Buildroot предоставляются только по общему доступному лицензионному соглашению.

Технические подробности см. в главе [19](#).

## Глава 14

# Помимо Buildroot

### 14.1 Загрузка с генерированных образов

#### 14.1.1 Загрузка NFS

Для загрузки по NFS включите корневую файловую систему tar в меню образов файловой системы.

После завершения сборки просто выполните следующие команды для настройки корневого каталога NFS:

```
sudo tar -xavf /путь/к/вых одному_каталогу/rootfs.tar -C /путь/к/nfs_root_dir
```

Не забудьте добавить этот путь в /etc/exports.

Затем вы можете выполнить загрузку NFS с вашего целевого устройства.

#### 14.1.2 Живой CD

Чтобы создать образ Live CD, включите опцию образа iso в меню Образы файловой системы. Обратите внимание, что эта опция доступна только для архитектур x86 и x86-64, а также если вы с обираете свое ядро с помощью Buildroot.

Вы можете создать образ Live CD с IsoLinux, Grub или Grub 2 в качестве загрузчика, но только IsoLinux поддерживает возможность использования этого образа как Live CD и Live USB (с помощью опции «Создать гибридный образ»).

Вы можете протестировать свой образ Live CD с помощью QEMU:

```
qemu-system-i386 -cdrom вывод/images/rootfs.iso9660
```

Или используйте его как образ жесткого диска, если это гибридный ISO:

```
qemu-system-i386 -hda вывод/images/rootfs.iso9660
```

Его можно легкозаписать на USB-накопитель с помощью dd:

```
dd если=вывод/images/rootfs.iso9660 из=/dev/sdb
```

### 14.2 Chroot

Если вы хотите выполнить chroot в сгенерированном образе, то вам следует знать несколько вещей:

- вам следует настроить новый корень из образа корневой файловой системы tar;
- либо выбранная архитектура совместима с вашей хост-машиной, либо вам следует использовать какой-либо двоичный файл qemu-\* и правильно настроить его в соответствии с вашими binfmt, чтобы иметь возможность запускать двоичные файлы, собранные для ядра, на вашей хост-машине;
- в настоящее время Buildroot не предоставляет host-qemu и binfmt, правильно собранные и настроенные для такого же использования.

## Час тъ 3

Руководство разработчика

# Глава 15

## Как работает Buildroot

Как упоминалось выше, Buildroot — это, по сути, набор Makefiles, которые загружают, настраивают и компилируют программное обеспечение с правильными параметрами. Он также включает патчи для различных пакетов программного обеспечения — в основном тех, которые задействованы в цепочке инструментов кросскомпиляции (gcc, binutils и uClibc).

По сути, существует один Makefile на пакет программного обеспечения и они называются расширением .mk. Makefiles разделены на множество различных частей.

- Каталог `toolchain/` содержит файлы Makefile и связанные с ними файлы для каждого пакета программного обеспечения с вложенным кросскомпиляцией. набор инструментов: binutils, gcc, gdb, kernel-headers и uClibc.
- Каталог `arch/` содержит определения для всех архитектур процессоров, которые поддерживаются Buildroot.
- Каталог `package/` содержит файлы Makefile и связанные с ними файлы для всех инструментов и библиотек пользовательского пространства, которые может использовать Buildroot. с компилировать и добавить в цепевую корневую файловую систему. Напакет приходит один подкаталог .
- Каталог `linux/` содержит файлы Makefile и связанные с ними файлы для ядра Linux.
- Каталог `boot/` содержит файлы Makefile и связанные с ними файлы для загрузчиков, поддерживаемых Buildroot.
- Каталог `system/` содержит поддержку системной интеграции, например, скелет цепевой файловой системы и выбор начальной загрузки.
- Каталог `fs/` содержит файлы Makefile и связанные с ними файлы для программного обеспечения с вложенным генерацией цепевой корневой файловой системы. изображение.

Каждый каталог содержит минимум 2 файла:

- `something.mk` — это Makefile, который загружает, настраивает, компилирует и устанавливает пакет `something`.
- `Config.in` — часть файла описания инструмента конфигурации. Он описывает опции, связанные с пакетом.

Основной Makefile выполняет следующие шаги (после завершения настройки):

- Создайте все необходимые каталоги: `staging`, `target`, `build` и т. д. в вых однократного каталога (по умолчанию `output/`, другое значение можно указать с помощью `O=`)
- Сгенерируйте цепевой набор инструментов. При использовании внутреннего набора инструментов это означает генерацию набора инструментов кросскомпиляции. При использовании внешнего набора инструментов это означает проверку функций внешнего набора инструментов и импорта в среду Buildroot.
- Сгенерировать все цели, перечисленные в переменной `TARGETS`. Эта переменная заполняется Makefile-ами в их отдельных компонентов. Генерация этих целей запускает компиляцию пакетов пользовательского пространства (библиотек, программ), ядра, загрузчика и генерацию образов корневой файловой системы, в зависимости от конфигурации.

# Глава 16

## Стиль кодирования

В целом, эти правила стиля кодирования призваны помочь вам добавлять новые файлы в Buildroot или рефакторить существующие.

Если вы немного измените какой-либо существующий файл, важно сюда заняться его файлом, чтобы вы могли:

- либо следовать потенциальному существующему стилю кодирования и использовать его в этом файле,
- или полностью переработать его, чтобы он соответствовал этим правилам.

### 16.1 Файл config.in

Файлы config.in содержат записи практически для всего, что можно настроить в Buildroot.

Запись имеет следующий шаблон:

```
конфигурация BR2_PACKAGE_LIBFOO
бул "libfoo"
зависит от BR2_PACKAGE_LIBBAZ выберите
BR2_PACKAGE_LIBBAR help Это комментарий,
который
объясняет, что такое libfoo. Текст с правкой должен быть заключен в обертку.
```

<http://foosoftware.org/libfoo/>

- Строки bool, depend on, select и help имеют отступ в одну табуляцию.
- Сам текст с правкой должен иметь отступ в одну табуляцию и два пробела.
- Текст с правкой должен быть разбит на 72 столбца, где табуляция считается языком, то есть в самом тексте должно быть 62 символов.

Файлы Config.in являются языком для инструмента конфигурации, используемого в Buildroot, который является обычным Kconfig. Для получения дополнительных сведений о языке Kconfig обратитесь к <http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt>.

### 16.2 Файл .mk

- Заголовок: Файл начинается с заголовка. Он содержит имя модуля желательно в нижнем регистре, заключенное между разделителями, сделанными из 80 хешей. После заголовка обязательно пустая строка

```
## ...
```

- Назначение: ис пользуйте = с одним пробелом перед ним и одним пробелом после него:

```
LIBFOO_VERSION = 1.0
LIBFOO_CONF_OPTS += --without-python-support
```

Не выравнивайте знаки =.

- Отсюда используйте только табуляции:

```
определить LIBFOO_REMOVE_DOC
$(RM) -r $(TARGET_DIR)/usr/share/libfoo/doc \
$(TARGET_DIR)/usr/share/man/man3/libfoo*
отпустить
```

Обратите внимание, что команды внутри блока определения всегда должны начинаться с символа табуляции, чтобы make распознавал их как команды.

- Не обязательная зависимость:

- Предпочитайте многострочный синтаксис.

ДА:

```
ifeq ($(BR2_PACKAGE_PYTHON3),y)
LIBFOO_CONF_OPTS += --with-python-support
LIBFOO_DEPENDENCIES += python3
еле
LIBFOO_CONF_OPTS += --without-python-support endif
```

НЕТ:

```
LIBFOO_CONF_OPTS += --with$(if $(BR2_PACKAGE_PYTHON3),out)-python-support
LIBFOO_DEPENDENCIES += $(if $(BR2_PACKAGE_PYTHON3),python3,
```

- Располагайте параметры конфигурации и зависимости близко друг к другу.

- Необязательные услуги: сюда раняйте определение услуги и ее назначение в одном блоке if.

ДА:

```
ifneq ($(BR2_LIBFOO_INSTALL_DATA),y) определить
LIBFOO_REMOVE_DATA
$(RM) -r $(TARGET_DIR)/usr/share/libfoo/data
отпустить
LIBFOO_POST_INSTALL_TARGET_HOOKS += LIBFOO_REMOVE_DATA
эндиф
```

НЕТ:

```
определить LIBFOO_REMOVE_DATA
$(RM) -r $(TARGET_DIR)/usr/share/libfoo/data
отпустить

ifneq ($(BR2_LIBFOO_INSTALL_DATA),y)
LIBFOO_POST_INSTALL_TARGET_HOOKS += LIBFOO_REMOVE_DATA конец_файла
```

## 16.3 Файл genimage.cfg

Файлы genimage.cfg содержат макет выходных изображений, который утилита genimage использует для создания конечного файла .img.

Ниже приведен пример:

```
изображение efi-part.vfat {
    vfat {
        файл EFI {
            изображение = "efi-частъ/EFI"
        }

        Файл Изображение
        { image = "Изображение"
        }

    }

    размер = 32M
}

изображение sdimage.img { hdimage
{}


раздел u-boot { image = "efi-
part.vfat" смещение = 8K

}

корень раздела { image =
"rootfs.ext2" size = 512M

}
}
```

- Каждый раздел (например, hdimage, vfat и т. д.) должен иметь отступ в одну табуляцию.
  - Каждый файл или другой подузел должен иметь отступ в два символа табуляции.
  - Каждый узел (раздел, раздел, файл, подузел) должен иметь открытую фигурную скобку на той же строке имени узла, в то время как закрывающая скобка должна быть на новой строке, и после нее должна быть добавлена новая строка, за исключением последнего узла. То же самое касается его опций, например, option size =.
  - Каждый параметр (например, изображение, смещение, размер) должен иметь назначение = на один пробел от него и на один пробел от значения указано.
  - Имя файла должно начинаться как минимум с префиксом genimage и иметь расширение .cfg, чтобы его можно было легко узнать.
  - Допустимые обозначения для параметров смещения и размера G, M, K (не k). Если невозможно выразить точное количество байт с помощью обозначений выше, используйте шестнадцатеричный префикс 0x или, в крайнем случае, количество байт в комментариях вместо G, M, K и используйте GB, MB, KB (не kb).
  - Для разделов GPT значение partition-type=uuid должно быть U для системного раздела EFI (расширенное до c12a7328-f81f-1 для раммы genimage), F для раздела FAT (расширенное до ebd0a0a2-b9e5-4433-87c0-68b6b72699c7 для раммы genimage) или L для корневой файловой системы или других файловых систем (расширенное до 0fc63daf-8483-4772-8e79-3d69d8477de4 для раммы genimage).
- Несмотря на то, что L является значением по умолчанию для genimage, мы предпочитаем явно указывать его в наших файлах genimage.cfg. Наконец, эти сокращения следует использовать без двойных кавычек, например partition-type=uuid = U. Если явно указан GUID, следующий за ним следует использовать точные буквы.

Файлы genimage.cfg являются языком для инструмента genimage, используемого в Buildroot для генерации конечного образа (т. е. sdcard.img). Более подробную информацию о языке genimage можно найти по адресу <https://github.com/pengutronix/genimage/blob/master/README.rst>.

## 16.4 Документация

В документации используется `asciidoc` формат.

Более подробную информацию о синтаксисе `asciidoc` можно найти по адресу <https://asciidoc-py.github.io/userguide.html>.

## 16.5 Поддержка криптов

Некоторые скрипты в каталогах `support/` и `utils/` написаны на Python и должны соответствовать [руководству по стилю PEP8 для кода Python](#).

# Глава 17

## Добавление поддержки для определенной платы

Buildroot содержит базовые конфигурации для нескольких общих типовых аппаратных плат, так что пользователи такой платы могут легким образом построить систему, которая заранее работает. Вы также можете добавить поддержку других плат в Buildroot.

Для этого вам нужно создать обычную конфигурацию Buildroot, которая сдает базовую систему для обнаружения (внутреннюю) почту инструментов, ядро, загрузчик, файловую систему и прочее пользовательское программное обеспечение BusyBox-only. Не следует выбирать какой-либо конкретный пакет: конфигурация должна быть минимальной и должна сдавать только рабочую базовую систему BusyBox для целевой платформы. Конечно, вы можете использовать более сложные конфигурации для своих внутренних проектов, но проект Buildroot будет интегрировать только базовые конфигурации платы. Это связано с тем, что выбор пакетов сильно зависит от приложения.

Как только у вас будет известна рабочая конфигурация, запустите make savedefconfig. Это сгенерирует минимальный файл defconfig в корне исходного дерева Buildroot. Переместите этот файл в каталог configs/ и переименуйте его в <boardname>.defconfig.

Всегда используйте фиксированные версии или эти коммиты для разных компонентов, а не "последнюю" версию. Например, установите BR2\_LINUX\_KERN и BR2\_LINUX\_KERNEL\_CUSTOM\_VERSION\_VALUE для версии ядра, с которой вы тестировали. Если вы используете buildroot toolchain BR2\_TOOLCHAIN\_BUILDROOT (который используется по умолчанию), дополнительно убедитесь, что используете заголовки ядра (BR2\_KERNEL\_HEADERS\_AS\_KERNEL, который также используется по умолчанию) и установите пользовательскую серию заголовков ядра в соответствии с вашей версией ядра (BR2\_PACKAGE\_HOST\_LINUX\_HEADERS\_CUSTOM\_\*).

Рекомендуется использовать как можно больше версий ядра Linux и загрузчиков, а также как можно больше конфигураций ядра и загрузчика по умолчанию. Если они не подходят для вашей платы или не существует конфигурации по умолчанию, мы рекомендуем вам отправлять исправления в соответствующие проекты по открытым.

Однако в то же время вам может потребоваться сначала сконфигурить конфигурацию ядра или загрузчика, а также исправления специфичные для вашей целевой платформы. Для этого создайте каталог board/<manufacturer> и подкаталог board/<manufacturer>/<boardname>.

Затем вы можете сначала скопировать конфигурации в эти каталогах и скопировать их из основной конфигурации Buildroot.

Более подробную информацию см. в Главе 9.

Перед отправкой патчей для новых докеров рекомендуется протестировать их, построив с использованием последнего контейнера gitlab-CI docker. Для этого используйте скрипт utils/docker-run внутри него выполните следующие команды:

```
$ make <имя_платы>.defconfig $ make
```

По умолчанию разработчики Buildroot используют официальный образ, размещенный в [репозитории gitlab.com](#). Это должно быть удобно для большинства случаев использования. Если вы все еще хотите создать свой собственный образ docker, вы можете взять его на основе официального образа в качестве директивы FROM вашего собственного Dockerfile:

```
Из registry.gitlab.com/buildroot.org/buildroot/base:ГГГММДДЧЧММ  
БЕГАТЬ ...  
КОПИРОВАТЬ ...
```

Текущую версию YYYYMMDD.HHMM можно найти в файле .gitlab-ci.yml в верхней части исходного дерева Buildroot; все предыдущие версии также перечислены в вышеупомянутом репозитории.

# Глава 18

## Добавление новых пакетов в Buildroot

В этом разделе описывается, как можно интегрировать новые пакеты (библиотеки или приложения пользовательского пространства) в Buildroot. Он также показывает, как интегрируются существующие пакеты, что необязательно для исправления проблем или настройки их конфигурации.

При добавлении нового пакета обязательно протестируйте его в различных условиях (см. раздел [18.25.3](#)), а также проверьте его стиль кодирования (см. раздел [18.25.2](#)).

### 18.1 Каталог пакетов

Прежде всего, создайте каталог в каталоге пакета для вашего программного обеспечения, например, libfoo.

Некоторые пакеты были сгруппированы по темам в подкаталогах: x11r7, qt5 и gstreamer. Если ваш пакет попадает в одну из этих категорий, то создайте каталога для пакета в них. Однако новые подкаталоги не приветствуются.

### 18.2 Файлы конфигурации

Чтобы пакет отображался в инструменте конфигурации, вам необходимо создать файл Config в каталоге вашего пакета. Существует два типа: Config.in и Config.in.host.

#### 18.2.1 Файл config.in

Для пакетов, используемых нацели, создайте файл с именем Config.in. Этот файл будет содержать описание опций, связанных с нашим программным обеспечением libfoo, которое будет использоваться и отображаться в инструменте конфигурации. Он должен в основном содержать:

```
конфигурация BR2_PACKAGE_LIBFOO
бул "libfoo"
help
Это комментарий, который объясняет, что такое libfoo. Текст с правкой должен быть перенесен.
```

<http://foosoftware.org/libfoo/>

Строка bool, следующие за ней метаданные о параметрах конфигурации должны иметь формат в одну табуляцию. Сам текст с правкой должен иметь формат в одну табуляцию и два пробела, строки должны быть перенесены так, чтобы вместе с табуляцией 72 символами, где табуляция считается языком, то есть 62 символом в самом тексте. Текст с правкой должен содержать URL-адрес проекта после пустой строки.

В соответствии со спецификацией для Buildroot, порядок атрибутов следующий:

1. Тип параметра: bool, string... с приложением

2. При необходимости, значение(я) по умолчанию
3. Любой зависит от цели в зависимости от формы
4. Любой зависит от цепочки инструментов зависит от формы
5. Любой зависит от других пакетов в зависимости от формы
6. Любой зависит от выбранной формы
7. Ключевое слово и текст с правки.

Вы можете добавить другие подопции в оператор if BR2\_PACKAGE\_LIBFOO...endif для настройки определенных вещей в вашем программном обеспечении. Вы можете посмотреть примеры в других пакетах. Синтаксис файла Config.in такой же, как и для файла ядра Kconfig. Документация по этому синтаксису доступна по адресу <http://kernel.org/doc/Documentation/kbuild/kconfig-language.txt>

Наконец, вам нужно добавить ваш новый libfoo/Config.in в package/Config.in (или в подкаталог категорий, если вы решили поместить свой пакет в одну из существующих категорий). Файлы, включенные туда, отсортированы в алфавитном порядке по категориям и НЕ должны сдерживать ничего, кроме имени пакета.

источник "package/libfoo/Config.in"

## 18.2.2 Файл config.in.host

Некоторые пакеты также должны быть собраны для хост-системы. Здесь есть два варианта:

- Пакет хоста требуется только для удовлетворения зависимостей времени сборки одного или нескольких целевых пакетов. В этом случае добавьте host-foo в переменную BAR\_DEPENDENCIES целевого пакета. Файл Config.in.host создавать не следует.
- Пакет хоста должен быть явно выбран пользователем из меню конфигуратора. В этом случае создайте Config.in.host файл для этого пакета хоста:

```
конфигуратор BR2_PACKAGE_HOST_FOO
bool "host foo"
help
    Это комментарий, который объясняет, что такое foo для хоста.

    http://foosoftware.org/foo/
```

Действительны те же стили кодирования и параметры, что и для файла Config.in.

Наконец, вам нужно добавить ваш новый libfoo/Config.in.host в package/Config.in.host. Файлы, включенные туда, отсортированы в алфавитном порядке и НЕ должны сдерживать ничего, кроме имени пакета.

источник "package/foo/Config.in.host"

После этого пакет хоста будет добавлен в меню «Утилиты хоста».

## 18.2.3 Выбор зависимостей или выбрать

Файл Config.in вашего пакета также должен гарантировать, что зависимости включены. Обычно Buildroot использует следующие правила:

- Используйте тип зависимости select для зависимостей от библиотек. Эти зависимости обычно не очевидны, поэтому имеет смысл, чтобы система kconfig обес печивала выбор зависимостей. Например, пакет libgtk2 использует select BR2\_PACKAGE\_LIBGLIB2, чтобы убедиться, что эта библиотека также включена. Ключевое слово select выражает зависимость с обратной семантикой.

- Используйте зависимость типа `depends on`, когда пользователю действительно нужно знать о зависимости. Обычно Buildroot использует этот тип зависимости для зависимостей от целевой архитектуры, поддержки MMU и параметров epoch инструментов (см. раздел 18.2.4) или для зависимостей от «больших» вендоров, таких как система X.org. Ключевое слово `depends on` выражает зависимость с прямой семантикой.

**Примечание** Текущая проблема с языком kconfig заключается в том, что эти две семантики зависимости не связаны между собой. Поэтому может быть возможным выбрать пакет, у которого одна из его зависимостей/требований не выполнена.

Пример иллюстрирует использование `select` и `depend on`.

```
конфигурация BR2_PACKAGE_RRDTOOL
```

```
бул "rrdtool"
зависит от BR2_USE_WCHAR выберите
BR2_PACKAGE_FREETYPE выберите BR2_PACKAGE_LIBART
выберите BR2_PACKAGE_LIBPNG выберите
BR2_PACKAGE_ZLIB help RRDtool — это
высокопроизводительная система
```

регистрации данных и построения графиков с открытым исходным кодом для временных рядов.

<http://oss.oetiker.ch/rrdtool/>

комментарий "rrdtool нуждается в epochке инструментов с wchar"

```
зависит от !BR2_USE_WCHAR
```

Обратите внимание, что эти два типа зависимости являются транзитивными только по отношению к зависимостям того же вида.

В следующем примере это означает:

```
конфиг BR2_PACKAGE_A bool "Пакет A"
```

```
config BR2_PACKAGE_B bool "Пакет B"
```

```
зависит от BR2_PACKAGE_A
```

```
config BR2_PACKAGE_C bool "Пакет C"
```

```
зависит от BR2_PACKAGE_B
```

```
config BR2_PACKAGE_D bool "Пакет D"
```

```
select BR2_PACKAGE_B
```

```
config BR2_PACKAGE_E bool "Пакет E"
```

```
select BR2_PACKAGE_D
```

- Выбор пакета C будет виден, если выбран пакет B, который, в свою очередь, виден только в том случае, если выбран пакет A. Выбрано.
- Выбор пакета E выберет пакет D, который выберет пакет B, он не будет проверять зависимости. Пакет B, поэтому пакет A не будет выбран.
- Поскольку пакет B выбран, а пакет A — нет, это нарушает зависимость пакета B от пакета A. Поэтому в такой ситуации транзитивную зависимость необходимо добавить явно:

```
config BR2_PACKAGE_D bool "Пакет D"
```

```
зависит от BR2_PACKAGE_A
```

```
выберите BR2_PACKAGE_B
```

```
config BR2_PACKAGE_E bool
    "Пакет E" зависит от
    BR2_PACKAGE_A select BR2_PACKAGE_D
```

В целом, для зависимостей библиотек и пакетов предпочтение следует отдавать `select`.

Обратите внимание, что такие зависимости гарантируют, что опция зависит от той же самой библиотеки, но не обязательно будет сопровождаться вашим пакетом. Для этого зависимость должна быть выражена в файле `.mk` пакета.

Дополнительные сведения о форматировании: см. [Стиль кодирования](#)

#### 18.2.4 Зависимости от параметров цели и цепочки инструментов

Многие пакеты зависят от определенных опций цепочки инструментов: выбор библиотеки C, поддержка C++, поддержка потоков, поддержка RPC, поддержка wchar или поддержка динамических библиотек. Некоторые пакеты могут быть построены только на определенных целевых архитектурах или если в процессе сборки используется MMU.

Эти зависимости должны быть выражены с помощью соответствующих зависимостей от операторов в файле `Config.in`. Кроме того, для зависимостей от опций цепочки инструментов должен отображаться комментарий, когда опция не включена, чтобы пользователь знал, почему пакет недоступен. Зависимости от целевой архитектуры или поддержки MMU не должны быть видны в комментарии: поскольку маловероятно, что пользователь сможет свободно выбрать другую цель, нет смысла явно показывать эти зависимости.

Комментарий должен быть видимым только в том случае, если сама опция конфигурации будет видимой при выполнении зависимости от опции `toolchain`. Это означает, что все остальные зависимости пакета (включая зависимости от целевой архитектуры и поддержки MMU) должны быть повторены в определении комментария. Чтобы было понятно, оператор `depend on` для этих опций `non-toolchain` должен быть отделен от оператора `depend on` для опции `toolchain`. Если в том же файле (обычно в основном пакете) есть зависимость от опции конфигурации, предпочтительнее иметь глобальную конфигурацию `if ... endif`, а не повторять оператор `depend on` для комментария и других опций конфигурации.

Общий формат комментария зависимости для пакета `foo`:

```
foo нужен набор инструментов с featA, featB, featC
```

например:

```
mpd нужен набор инструментов с C++, потоками, wchar
```

или

```
crda нужен набор инструментов с потоками
```

Обратите внимание, что этот текст намеренно сделан кратким, чтобы он поместился на 80-символьном терминале.

Вот та часть этого раздела, перечислены различные параметры целей и цепочки инструментов, соответствующие символам конфигурации, от которых они зависят, и текст, который следует использовать в комментарии.

- Целевая архитектура
  - Символ зависимости: `BR2_powerpc`, `BR2_mips`, ... (см. `arch/Config.in`)
  - Стока комментария комментарий не добавляется
- Поддержка MMU
  - Символ зависимости: `BR2_USE_MMU`
  - Стока комментария комментарий не добавляется

- Встроенные функции Gcc \_sync\* используются для атомарных операций. Они доступны в вариантах, работающих с 1 байтом, 2 байтами, 4 байтами и 8 байтами. Поскольку разные архитектуры поддерживают атомарные операции с разными размерами, для каждого размера доступен один символ зависимости:
  - Символ зависимости: BR2\_TOOLCHAIN\_HAS\_SYNC\_1 для 1 байта, BR2\_TOOLCHAIN\_HAS\_SYNC\_2 для 2 байтов, BR2\_TOOLC для 4 байтов, BR2\_TOOLCHAIN\_HAS\_SYNC\_8 для 8 байтов.
  - Стока комментария комментарий не добавляется
- Встроенные функции Gcc \_atomic\*, используемые для атомарных операций.
  - Символ зависимости: BR2\_TOOLCHAIN\_HAS\_ATOMIC.
  - Стока комментария комментарий не добавляется
- Заголовки ядра
  - Символ зависимости: BR2\_TOOLCHAIN\_HEADERS\_AT\_LEAST\_X\_Y, (замените X\_Y на правильную версию, см. toolchain/ - Стока комментария headers >= XY и/или headers <= XY (замените XY на правильную версию)
- Версия GCC
  - Символ зависимости: BR2\_TOOLCHAIN\_GCC\_AT\_LEAST\_X\_Y, (замените X\_Y на правильную версию, см. toolchain/Conf - Стока комментария gcc >= XY и/или gcc <= XY (замените XY на правильную версию)
- Хост-версия GCC
  - Символ зависимости: BR2\_HOST\_GCC\_AT\_LEAST\_X\_Y, (замените X\_Y на правильную версию, см. Config.in)
  - Стока комментария комментарий не добавляется
  - Обратите внимание, что обычно минимальную версию хоста GCC имеет не сам пакет, а хост-пакет, на котором он установлен. зависит от.
- Библиотека Си
  - Символ зависимости: BR2\_TOOLCHAIN\_USES\_GLIBC, BR2\_TOOLCHAIN\_USES\_MUSL, BR2\_TOOLCHAIN\_USES\_UCLIB
  - Стока комментария для библиотеки С используется немногодругой текст комментария foo требуется набор инструментов glibc, или foo нужен набор инструментов glibc с C++
- Поддержка C++
  - Символ зависимости: BR2\_INSTALL\_LIBSTDCPP
  - Стока комментария C++
- Поддержка Д
  - Символ зависимости: BR2\_TOOLCHAIN\_HAS\_DLNG
  - Стока комментария Dlang
- Поддержка Фортрана
  - Символ зависимости: BR2\_TOOLCHAIN\_HAS\_FORTRAN
  - Стока комментария fortran
- Поддержка нитей
  - Символ зависимости: BR2\_TOOLCHAIN\_HAS\_THREADS
  - Стока комментария потоки (если только не требуется я также BR2\_TOOLCHAIN\_HAS\_THREADS\_NPTL, в этом случае следует указать досстаточно только NPTL)

- Поддержка потоков NPTL
  - Символ зависит от: BR2\_TOOLCHAIN\_HAS\_THREADS\_NPTL
  - Стока комментария NPTL
- Поддержка RPC
  - Символ зависит от: BR2\_TOOLCHAIN\_HAS\_NATIVE\_RPC
  - Стока комментария RPC
- поддержка wchar
  - Символ зависит от: BR2\_USE\_WCHAR
  - Стока комментария wchar
- динамическая библиотека
  - Символ зависит от: !BR2\_STATIC\_LIBS – Стока комментария динамической библиотеки

### 18.2.5 Зависимости от ядра Linux, с обранном о buildroot

Для сборки некоторых пакетов требуется ядро Linux с помощью buildroot. Обычно это модули ядра или прошивки. В файле Config.in следует добавить комментарий, чтобы выразить эту зависимость, аналогично зависимости от опциональной toolchain. Общий формат:

Для сборки foo необъодимо ядро Linux

Если есть зависимость как от параметров цепочки инструментов, так и от ядра Linux, ис пользуйте следующий формат:

Для сборки foo требуется набор инструментов с featA, featB, featC и ядром Linux

### 18.2.6 Зависимости от управления udev /dev

Если пакету требуется управление udev /dev, он должен зависеть от символа BR2\_PACKAGE\_HAS\_UDEV, и следует добавить следующий комментарий:

foo требуется управление udev /dev

Если есть зависимость как от параметров цепочки инструментов, так и от управления udev /dev, ис пользуйте следующий формат:

foo требуется управление udev /dev и набор инструментов с featA, featB, featC

### 18.2.7 Зависимости от функций, предоставляемых виртуальными пакетами

Некоторые функции могут предоставляться более чем одним пакетом, например, библиотеками OpenGL.

Более подробную информацию о виртуальных пакетах см. в разделе [18.12](#).

## 18.3 Файл .mk

Наконец, вот самая ложная часть. Создайте файл с именем libfoo.mk. Он описывает, как пакет должен быть загружен, настроен, установлен и т. д.

В зависимости от типа пакета файл .mk должен быть написан по-разному, с использованием различных инфраструктур:

- Makefiles для универсальных пакетов (не использующих autotools или CMake): они основаны на инфраструктуре, поэтому очевидно, что используются ядрами пакетов на основе autotools, но требуют немногого больше работы от разработчика. Они определяют, что должно быть сделано для настройки, компиляции и установки пакета. Эта инфраструктура должна использовать ядра других пакетов, которые не используют autotools в качестве своей системы сборки. В будущем могут быть написаны другие специализированные инфраструктуры для других систем сборки. Мы рассмотрим их в [руководстве](#) и [справочнике](#).
- Makefiles для программного обеспечения на основе autotools (autoconf, automake и т. д.): Мы предоставляем специальную инфраструктуру для таких пакетов, поскольку autotools — очень распространенная система сборки. Эта инфраструктура не обходимо использовать для новых пакетов, которые полагаются на autotools как на свою систему сборки. Мы рассмотрим их в [руководстве](#) и [справочнике](#).
- Makefiles для ПО на основе cmake: Мы предоставляем выделенную инфраструктуру для таких пакетов, поскольку CMake становится все более и более распространенной системой сборки и имеет стандартизированное поведение. Эта инфраструктура должна использовать ядра новых пакетов, которые полагаются на CMake. Мы рассмотрим их в [руководстве](#) и [справочнике](#).
- Makefiles для модулей Python: у нас есть выделенная инфраструктура для модулей Python, которые используют механизмы flit, hatch, pep517, poetry setuptools, setuptools-rust или maturin. Мы рассмотрим их в [руководстве](#) и [справочнике](#).
- Makefiles для модулей Lua: у нас есть специальная инфраструктура для модулей Lua, доступная на веб-сайте LuaRocks. Мы рассмотрим их в [учебном пособии](#) и [справочном материале](#).

Дополнительные сведения о форматировании: см. [правила написания](#).

## 18.4 Файл .hash

Когда это возможно, вы должны добавить третий файл с именем libfoo.hash, который содержит эши загруженных файлов для пакета libfoo. Единственная причина, по которой не следует добавлять файл .hash, — это когда проверка эши невозможна из-за ошибки загрузки пакета.

Если пакет имеет возможность выбора версии, то эш-файл может храниться в подкаталоге, названном по версии, например, package/libfoo/1.2.3/libfoo.hash. Это особенно важно, если разные версии имеют различные условия лицензирования, но они хранятся в одном файле. В противном случае эш-файл должен оставаться в каталоге пакета.

Хеш-коды, хранящиеся в этом файле, используются для проверки целостности загруженных файлов и файлов лицензии.

Формат этого файла — одна строка для каждого файла, хеш которой проверяется, каждая строка содержит три поля, разделенные двумя пробелами:

- тип эши, один из:
  - md5, sha1, sha224, sha256, sha384, sha512
- хэш файла:
  - для md5, 32 шестнадцатеричных символов
  - для sha1, 40 шестнадцатеричных символов
  - для sha224, 56 шестнадцатеричных символов
  - для sha256, 64 шестнадцатеричных символов
  - для sha384, 96 шестнадцатеричных символов
  - для sha512, 128 шестнадцатеричных символов

- имя файла:

- для исходного архива базовое имя файла без какого-либо компонента каталога, - для файла лицензии: путь, как он указан в FOO\_LICENSE\_FILES.

Строки, начинающиеся с символом #, считаются комментариями и игнорируются. Пустые строки игнорируются.

Для одного файла может быть более одного хеша, каждый на своей строке. В этом случае все хеши должны совпадать.

Примечание. В идеале хеш, хранящийся в этом файле, должны соответствовать хешам, опубликованным upstream, например, на их веб-сайте, в электронном письме. Если upstream не предоставляет более одного отпечатка хеша (например, sha1 и sha512), то лучше всего добавить все эти анонсирующие хеши в файл .hash. Если предоставляет никаких хешей или предоставляет только один md5, то вычислите по крайней мере один сильный хеш с помощью (предпочтительно sha256, но не md5) и укажите это в строке комментария над хешами.

Примечание. Хеш для файлов лицензий используется для обнаружения изменения лицензии при повышении версии пакета. Хеш проверяется во время выполненияцеликом make legal-info. Для пакета с несколькими версиями (например, Qt5) создайте файл хеша в подкаталоге <packageversion> этого пакета (см. также раздел 19.2).

В примере ниже определяются хеши sha1 и sha256, опубликованные выше с точки зрения разработчиками для нового tar-архива libfoo-1.2.3.tar.bz2, хеш md5 из вышеупомянутого разработчика и локально вычисленные хеши sha256 для двоичного BLOB-объекта, хеш sha256 для загруженного патча и архива без хеша:

```
# Хеш из: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.{sha1,sha256}:
sha1 486fb55c3efa71148fe07895fd713ea3a5ae343a libfoo-1.2.3.tar.bz2
sha256 efc48103cc3bcb06bda6a781532d12701eb081ad83e8f90004b39ab81b65d4369 libfoo-1.2.3.tar.

632

# md5 из: http://www.foosoftware.org/download/libfoo-1.2.3.tar.bz2.md5, sha256 локально
вычислено
md5 2d608f3c318c6b7557d551a5a09314f03452f1a1 libfoo-data.bin
sha256 01ba4719c80b6fe911b091a7c05124b64eece964e09c058ef8f9805daca546b libfoo-data.bin

# Локально вычисляется sha256
ff52101fb90bbfc3fe9475e425688c660f46216d7e751c4bbdb1dc85cdccacb9 libfoo-fix-blabla
.патч

# Хеш для файлов лицензий:
sha256 a45a845012742796534f7e91fe623262ccfb99460a2bd04015bd28d66fba95b8 КОПИРОВАНИЕ
sha256 01b1f9f2c8ee648a7a596a1abe8aa4ed7899b1c9e5551bda06da6e422b04aa55 doc/КОПИРОВАНИЕ.LGPL
```

Если файл .hash присутствует и содержит один или несколько хешей для загруженного файла, хеш(и), вычисленные Buildroot (после загрузки), должны совпадать с хешем(ами), со временем в файле .hash. Если один или несколько хешей не совпадают, Buildroot считает это ошибкой, удаляет загруженный файл и прерывает работу.

Если файл .hash присутствует, но не содержит хеша для загруженного файла, Buildroot считает это ошибкой и прерывает работу.

Однако загруженный файл остается в каталоге загрузки, поскольку это обычно указывает на то, что файл .hash неверен, но загруженный файл, вероятно, в порядке.

В настоящее время хеш проверяется для файлов, полученных с http/ftp-серверов, репозиториев Git или svn, файлов, скопированных с помощью scp, и локальных файлов. Хеш не проверяется для других систем контроля версий (таких как CVS, mercurial), поскольку Buildroot в настоящее время не генерирует воспроизводимые tarballs, когда исходный код извлекается из таких систем контроля версий.

Кроме того, для пакетов, для которых можно указать пользовательскую версию (например, пользовательскую строку версии, URL-адрес удаленного tarball или местоположение репозитория VCS и набор изменений), Buildroot не может переносить хеш для них. Однако можно предоставить с письмом дополнительных хешей, которые могут покрыть такие случаи.

Хеш следует добавлять только в файлы .hash для файлов, которые гарантированно стабильны. Например, патчи, автоматически сконструированные Github, не гарантированно стабильны, поэтому их хеш может со временем меняться. Такие патчи не следует загружать, вместе с тем добавлять локально в папку пакета.

Если файл .hash отсутствует, то проверка вообще не производится.

## 18.5 Стартовый скрипт SNNfoo

Пакеты, предстающие с системным демоном, обычно должны быть каким-то образом запущены при загрузке. Buildroot предоставляет яс поддержкой нескольких систем инициализации, некоторые из них считаются системами первого уровня (см. раздел 6.3), другие же доступны, но не имеют такого же уровня интеграции. В идеале все пакеты, представляющие с системный демон, должны предоставлять сценарий для BusyBox/SysV init и файл journala systemd.

Для обеспечения единства сценария сценарий должен соответствовать стилю и ставу, указанным в правочнике: package/busybox/S01syslogd.

Ниже показан аннотированный пример этого стиля. Для файлов journala systemd нет определенного стиля кодирования, но если пакет предоставляет яс с своим собственным файлом journala, он предпочтительнее, чем специальный для buildroot, если он совместим с buildroot.

Имя сценария скрипта состоит из SNN и имени демона. NN — это номер порядка запуска, который необходимо обязательно выбирать. Например, программа, требующая подключения к сети, не должна запускаться ядо S40network. Скрипты запускаются в алфавитном порядке, поэтому S01syslogd запускается ядо S01watchdogd, а S02sysctl — после него.

```
#!/bin/sh

DAEMON="syslogd"
PIDFILE="/var/run/$DAEMON.pid"

SYSLOGD_ARGS=""

# shellcheck source=/dev/null [-r "/etc/default/
$DAEMON"] && . "/etc/default/$DAEMON"

# BusyBox' syslogd не создает pid-файл, поэтому передайте "-n" в командной строке # и используйте "--make-pidfile", чтобы указать
start-stop-daemon создать его. start() {
    printf 'Запуск %s: '$DAEMON' # shellcheck
    disable=SC2086 # нам нужно разделение слов start-stop-daemon --start --background --make-
    pidfile \--pidfile "$PIDFILE" --exec "/sbin/$DAEMON" \-- -n $SYSLOGD_ARGS

    статус=$?
    если [ "$status" -eq 0 ]; тогда
        echo "OK"
    иначе
        echo "FAIL"
    конец
    вернуть "$status"
}

stop() { printf
    'Остановка %s: '$DAEMON' start-stop-daemon --stop --
    pidfile "$PIDFILE" --exec "/sbin/$DAEMON" status=$?

    если [ "$status" -eq 0 ]; тогда
        echo "OK"
    иначе
        echo "FAIL" вернуть
        "$status"
    конец
    while start-stop-daemon --stop --test --quiet --pidfile "$PIDFILE" \--exec "/sbin/$DAEMON"; do sleep 0.1

    сделанный
    rm -f "$PIDFILE" возвращает
    "$status"
}

перезапуск() {
    остановленный
}
```

```

    начинать
}

случай "$1" в
    с тарт|стоп|перезапуск) "$1";;

(перезагрузить)
# Перезапустить, так как настоящей функции «перезагрузки» нет. перезапустить;;
*)

echo "Использование: $0 {start|stop|restart|reload}"
выход 1
если

```

Для ясности в сценариях следует использовать длинные варианты.

### 18.5.1 Наследование сценария запуска

Как стартовые скрипты, так и файлы юнитов могут получать аргументы командной строки из `/etc/default/foo`, где `foo` — имя демона, заданное в переменной `DAEMON`. В общем случае, если такой файл не существует, он не должен блокировать запуск демона, если только нет какого-либо специфического для него аргумента командной строки, требуемого демоном для запуска. Для стартовых скриптов `FOO_ARGS="-s -o -m -e -args"` может быть определено как значение по умолчанию в скрипте, и пользователь может переопределить его из `/etc/default/foo`.

### 18.5.2 Обработка PID-файла

Файл PID не нужен для отслеживания нового процесса службы. Как с ним обращаться зависит от того, создает ли служба свой собственный файл PID и удаляет ли она его при завершении работы.

- Если ваша служба создает свой собственный файл PID, вызовите демон в режиме переднего плана и используйте `start-stop-daemon --make-pidfile --background`, чтобы `start-stop-daemon` создал файл PID. См. пример `S01syslogd`:

```
start-stop-daemon --start --background --make-pidfile \ --pidfile "$PIDFILE" --exec "/sbin/$DAEMON"
\ -- -n $SYSLOGD_ARGS
```

- Если ваша служба создает свой собственный PID-файл, передайте параметр `--pidfile` как `start-stop-daemon`, так и самому демону (или задайте его в соответствии с вашим образом в файле конфигурации, в зависимости от того, что поддерживает демон), чтобы они сговаривали местонахождение PID-файла. См. пример `S45NetworkManager`:

```
start-stop-daemon --start --pidfile "$PIDFILE" \ --exec "/usr/sbin/$DAEMON" \ -- --pid-
file="$PIDFILE" $NETWORKMANAGER_ARGS
```

- Если ваша служба удаляет свой PID-файл при завершении работы, используйте циклическую проверку того, что PID-файл исчез при установке, см. `S45NetworkManager` например:

```
пока [ -f "$PIDFILE" ]; делать
    сон 0.1
сделанный
```

- Если ваша служба не удаляет свой PID-файл при завершении работы, используйте цикл с `start-stop-daemon`, проверяющий, работает ли служба, и удалите PID-файл после завершения процесса. См. пример `S01syslogd`:

```
while start-stop-daemon --stop --quiet --pidfile "$PIDFILE" \ --exec "/sbin/$DAEMON"; do sleep 0.1
сделанный
rm -f "$PIDFILE"
```

Обратите внимание на флаг `--test`, который сообщает start-stop-daemon, что на самом деле не нужно останавливать службу, а нужно проверить, возможно ли это сделать. Проверка завершается ошибкой, если служба не запущена.

### 18.5.3 Остановка службы

Функция `stop` должна проверить, что процесс с демоном действительно завершился, прежде чем вернуться в противном случае перезапуск может завершиться неудачей, поскольку новый экземпляр запущен вместо старого фактически остановился. Как это сделать, зависит от того, как обрабатывается файл PID для службы (см. выше). Рекомендуется явно добавлять `--exec "/sbin/$DAEMON"` в команду start-stop-daemon, чтобы гарантировать отправку сигналов в PID, соответствующий \$DAEMON.

### 18.5.4 Перезагрузка конфигурации и сервиса

Программы, которые поддерживают перезагрузку своей конфигурации каким-либо образом (например, SIGHUP), должны предоставлять функцию `reload()`, пока ожидая `stop()`. Команда `start-stop-daemon` поддерживает `--stop --signal HUP` для этого. При отправке сигналов таким образом, будь то SIGHUP или другое, обязательно используйте символические имена, а не номера сигналов. Номера сигналов могут различаться в зависимости от архитектуры ЦП, а имена также легче читать.

### 18.5.5 Коды возврата

Функции для каждого крипто-сервиса должны возвращать код возврата с ответствующим кодом для каждого start-stop-daemon. Последний из них должен быть кодом возврата с криптой в целом, чтобы обеспечить автоматическую проверку на успешность, например, при вызове каждого крипто-сервиса с криптами и другими криптами. Обратите внимание, что без явного возврата кода возврата по следней команды в крипте или функции установки я ее кодом возврата, поэтому явный возврат не всегда требуется.

### 18.5.6 Ведение журнала

Когда служба переходит в фоновый режим или это делает `start-stop-daemon -background`, `stdout` и `stderr` обычно закрываются и все сообщения журнала, которые с службой может туда записать, теряются. Если возможно, настройте службу на ведение журнала в `syslog` (предпочтительно) или в выделенный файл журнала.

## 18.6 Инфраструктура для пакетов с определенными системами с борками

Под пакетами с определенными системами с борками подразумеваем все пакеты, чьи системы с борками не является одной из стандартных, таких как autotools или CMake. Это обычно включает пакеты, чьи системы с борками основаны на написанных вручную Makefiles или с криптами оболочки.

### 18.6.1 Учебник по универсальному пакету

```

01: #####
02: #
03: # libfoo
04: #
05: #####
06:
07: LIBFOO_VERSION = 1.0.08:
LIBFOO_SOURCE = libfoo-$LIBFOO_VERSION.tar.gz 09: LIBFOO_SITE = http://
www.foosoftware.org/download 10: LIBFOO_LICENSE = GPL-3.0+
11: LIBFOO_LICENSE_FILES = COPYING
12: LIBFOO_INSTALL_STAGING = YES
13: LIBFOO_CONFIG_SCRIPTS = libfoo-config
14: LIBFOO_DEPENDENCIES = x86_64-libaa libbbb
15:

```

```

16: определить LIBFOO_BUILD_CMDS 17:
    $(MAKE) ${TARGET_CONFIGURE_OPTS} -C ${@D} в с е
18: уйти
19:
20: определить LIBFOO_INSTALL_STAGING_CMDS 21:
    $(INSTALL) -D -m 0755 ${@D}/libfoo.a ${STAGING_DIR}/usr/lib/libfoo.a $(INSTALL) -D -m 0644 ${@D}/foo.h ${STAGING_DIR}/usr/include/foo.h
22:     $(INSTALL) -D -m 0755 ${@D}/libfoo.so* ${STAGING_DIR}/usr/lib
23:
24: уйти
25:
26: определить LIBFOO_INSTALL_TARGET_CMDS
27:     $(INSTALL) -D -m 0755 ${@D}/libfoo.so* ${TARGET_DIR}/usr/lib $(INSTALL) -d -m 0755 ${TARGET_DIR}/etc/foo.d
28:
29: уйти
30:
31: определить LIBFOO_USERS 32: foo -1 libfoo
-1 * --- Демон LibFoo
33: конец
34:
35: определить LIBFOO_DEVICES 36:
    /dev/foo c 666 0 0 42 0 ---
36: уйти
37:
38:
39: определить LIBFOO_PERMISSIONS 40:
    /bin/foo f 4755 foo libfoo -----
40: конец
41:
42:
43: $(eval $generic-package))

```

Makefile начинается с строк 7-11 с метаданных: версия пакета (LIBFOO\_VERSION), имя tar-архива, содержащего пакет (LIBFOO\_SOURCE) (рекомендуется xz-ed tar-архив), местоположение в Интернете, откуда можно загрузить tar-архив (LIBFOO\_SITE), лицензия (LIBFOO\_LICENSE) и файл с текстом лицензии (LIBFOO\_LICENSE\_FILES).

Все переменные должны начинаться с одного и того же префикса, в данном случае LIBFOO\_. Этот префикс всегда дает представление о всей заглавной версии имени пакета (см. ниже, чтобы понять, где определяется имя пакета).

В строке 12 мы указываем, что этот пакет хочет установить что-то в промежуточное пространство. Это часто требуется для библиотек, поскольку они должны устанавливать файлы заголовков и другие файлы разработки в промежуточное пространство. Это гарантирует, что команды, перечисленные в переменной LIBFOO\_INSTALL\_STAGING\_CMDS, будут выполнены.

В строке 13 мы указываем, что необязательно внести некоторые изменения в некоторые файлы libfoo-config, которые были установлены во время фазы LIBFOO\_INSTALL. Эти файлы \*.config являются яис полными файлами скриптов оболочки, которые находятся в каталоге \${STAGING\_DIR}/usr/bin и выполняются ядром вместе с внешними пакетами для определения местоположения и флагов с включением этого конкретного пакета.

Проблема в том, что все эти файлы \*.config по умолчанию содержат неправильные флаги с включениями соответствующей системы, которые не подходят для компиляции.

Например: -I/usr/include вместо -I\${STAGING\_DIR}/usr/include или: -L/usr/lib вместо -L\${STAGING\_DIR}/usr/lib

Поэтому к этим скриптам применяется ямагия sed, чтобы заставить их выдавать правильные флаги. Аргумент, который следует передать LIBFOO\_CONFIG\_SCRIPTS, — это имя(я) файла скрипта(ов) оболочки, требующего изменения. В эти имена относятся \${STAGING\_DIR}/usr/bin, и при необходимости можно указать несколько имен.

Кроме того, скрипты, перечисленные в LIBFOO\_CONFIG\_SCRIPTS, удаляются из \${TARGET\_DIR}/usr/bin, поскольку они не нужны на целевой машине.

#### Пример 18.1 Скрипты конфигурации для пакета divine

divine устанавливает скрипты оболочки \${STAGING\_DIR}/usr/bin/divine-config.

Таким образом, если изменение будет следующим:

```
DIVINE_CONFIG_SCRIPTS = божественный конфигурационный пакет
```

**Пример 18.2 Скрипты конфигурации: пакет imagemagick: Пакет**

imagemagick устанавливается следующие скрипты: \$(STAGING\_DIR)/usr/bin/{Magick, Magick++, MagickCore, MagickWand, Wand}-config. Таким образом, исправление будет

следующим:

```
IMAGEMAGICK_CONFIG_SCRIPTS = \
    Magick-config Magick++-config \
    MagickCore-config MagickWand-config Wand-config
```

В строке 14 мы указываем список зависимостей, на которые опирается этот пакет. Эти зависимости перечислены в виде имен пакетов в нижнем регистре, которые могут быть пакетами для цели (без префикса host-) или пакетами для хоста (с префиксом host-). Buildroot гарантирует, что все эти пакеты будут собраны и установлены до того, как текущий пакет начнет свою конфигурацию.

Остальная часть Makefile, с строк 16..29, определяет, что должно быть сделано на различных этапах конфигурации пакета, компилиации и установки. LIBFOO\_BUILD\_CMDS сообщает, какие шаги следуют выполнить для сборки пакета. LIBFOO\_INSTALL\_STAGING\_ сообщает, какие шаги следуют выполнить для установки пакета в промежуточном пространстве. LIBFOO\_INSTALL\_TARGET\_CMDS сообщает, какие шаги следуют выполнить для установки пакета в целевом пространстве.

Все эти шаги основаны на переменной \$(@D), которая содержит каталог, в который был извлечен исходный код пакета.

В строках 31..33 мы определяем пользователя, который использует этим пакетом (например, для запуска демона без прав root) (LIBFOO\_USERS).

В строках 35..37 мы определяем файл установки устройства, используемый этим пакетом (LIBFOO\_DEVICES).

В строках 39..41 мы определяем разрешения, устанавливаемые для определенных файлов, устанавливаемых этим пакетом (LIBFOO\_PERMISSIONS).

Наконец, в строке 43 мы вызываем функцию generic-package, которая генерирует, в соответствии с определенными ранее переменными, весь код Makefile, необходимый для того, чтобы ваш пакет работал.

## 18.6.2 Создание общего пакета

Существует два варианта универсальной цели. Макрос generic-package используется для пакетов, которые должны быть скомпилированы для цели. Макрос host-generic-package используется для пакетов хоста, изначально скомпилированных для хоста. Можно вызывать их оба в одном файле .mk: один раз для создания правил для генерации целевого пакета и один раз для создания правил для генерации пакетов хоста.

```
$(eval $(generic-package)) $(eval $(host-generic-package))
```

Это может быть полезно, если компилияция целевого пакета требует установки некоторых инструментов хоста. Если имя пакета — libfoo, то имя пакета для цели — также libfoo, а имя пакета для хоста — host-libfoo. Эти имена следует использовать в переменных DEPENDENCIES других пакетов, если они зависят от libfoo или host-libfoo.

Вызов generic-package и/или host-generic-package макроса должен быть в конце файла .mk, после всех определений переменных. Вызов host-generic-package должен быть после вызова generic-package, если таковой имеется.

Для целевого пакета generic-package использует переменные, определенные в файле .mk и предваряемые заглавными буквами имени пакета: LIBFOO\_\*. host-generic-package использует переменные HOST\_LIBFOO\_\*. Для некоторых переменных, если префикс ная переменная HOST\_LIBFOO\_ не существует, инфраструктура пакета использует соответствующую переменную с префиксом LIBFOO\_. Это делается для переменных, которые, вероятно, будут иметь одинаковое значение как для целевого пакета, так и для хостового пакета. Подробности см. ниже.

Список переменных, которые можно задать в файле .mk для предоставления метаданных (предполагается что имя пакета — libfoo):

- LIBFOO\_VERSION, обязательно должен содержать версию пакета. Обратите внимание, что если HOST\_LIBFOO\_VERSION не существует, предполагается что он такой же, как LIBFOO\_VERSION. Это также может быть номер ревизии или тег для пакетов, которые извлекаются напрямую из их системы контроля версий. Примеры:

- версия для релизного tarball: LIBFOO\_VERSION = 0.1.2

- sha1 для дерева git: LIBFOO\_VERSION = cb9d6aa9429e838f0e54faa3d455bcbab5eef057 – тег для дерева git LIBFOO\_VERSION =

## v0.1.2

Примечание: ис пользование имени ветки как FOO\_VERSION не поддерживается поскольку оно не работает и не может работать так, как ожидалось у пользователем:

1. из-за локального кэширования Buildroot не будет повторно загружать репозиторий, поэтому люди, которые ожидают возможности следить за удаленным репозиторием, будут весьма удивлены и разочарованы; 2. поскольку две сборки никогда не могут быть полностью одновременными, удаленный репозиторий может в любое время получить новые коммиты в ветке, два пользователя ис пользующие один и тот же дерево Buildroot и собирающие одну и ту же конфигурацию, могут получить разный исходный код, что с делает с борку невоспроизводимой, и люди будут весьма удивлены и разочарованы.

- LIBFOO\_SOURCE может содержать имя tarball пакета, которое Buildroot будет использовать для загрузки tarball с LIBFOO\_SITE. Если HOST\_LIBFOO\_SOURCE не указан, по умолчанию ис пользуется LIBFOO\_SOURCE. Если ничего не указано, то предполагается, что значение равно libfoo-\$(LIBFOO\_VERSION).tar.gz.

Пример: LIBFOO\_SOURCE = foobar-\$(LIBFOO\_VERSION).tar.bz2

- LIBFOO\_PATCH может содержать разделенный пробелами список имен файлов патчей, которые Buildroot загрузит и применит к исходному коду пакета. Если запись содержит ::/, то Buildroot предположит, что это полный URL, и загрузит патч из этого места.

В противном случае Buildroot будет считать, что патч должен быть загружен с LIBFOO\_SITE. Если HOST\_LIBFOO\_PATCH не указан, по умолчанию ис пользуется LIBFOO\_PATCH. Обратите внимание, что патчи, включенные в сам Buildroot, ис пользуют другим механизмом: все файлы формы \*.patch, присутствующие в каталоге пакета внутри Buildroot, будут применены к пакету после извлечения (см. [исправление пакета](#)). Наконец, патчи, перечисленные в переменной LIBFOO\_PATCH, применяются к ядру патчей, хранящимся в каталоге пакета Buildroot.

- LIBFOO\_SITE предоставляет место расположение пакета, которое может быть URL-адресом или путем локальной файловой системы. HTTP, FTP и SCP являются поддерживаемыми типами URL для извлечения tar-архивов пакетов. В этих случаях не включайте завершающую косую черту: она будет добавлена Buildroot между каталогом и именем файла по мере необходимости. Git, Subversion, Mercurial и Bazaar являются поддерживаемыми типами URL для извлечения пакетов напрямую из исходного управления исходным кодом. Существует вспомогательная функция упрощенного загрузки исходных tar-архивов с GitHub (подробнее см. в разделе [18.25.4](#)). Путь файловой системы может ис пользоваться ядру указания либо tar-архива, либо каталога, содержащего исходный код пакета. Подробнее о том, как работает извлечение, см. в разделе LIBFOO\_SITE\_METHOD ниже.

Обратите внимание, что URL-адрес с SCP должны иметь вид scp://[user@]host:filepath, а путь к файлу указывается относительно домашнего каталога пользователя поэтому для абсолютных путей может потребоваться добавить к пути косую черту: scp://[user@]host:/absolutepath. То же самое касается URL-адресов SFTP.

Если HOST\_LIBFOO\_SITE не указан, по умолчанию ис пользуется LIBFOO\_SITE. Примеры:

LIBFOO\_SITE=http://www.libfoosoftware.org/libfoo LIBFOO\_SITE=http://svn.xiph.org/

trunk/Tremor LIBFOO\_SITE=/opt/software/libfoo.tar.gz LIBFOO\_SITE=\$(TOPDIR)../

src/libfoo

- LIBFOO\_DL\_OPTS — это разделенный пробелами список дополнительных опций для передачи загрузчику. Полезно для извлечения документов с проверкой на стороне сервера инноваций и паролей пользователей или для ис пользования прокси. Поддерживаются все методы загрузки, допустимые для LIBFOO\_SITE\_METHOD; допустимые параметры зависят от метода загрузки (см. страницу руководства для соответствующих утилит загрузки).

- LIBFOO\_EXTRA\_DOWNLOADS — это разделенный пробелами список дополнительных файлов, которые Buildroot должен загрузить. Если запись с содержит ::/, то Buildroot предположит, что это полный URL, и загрузит файл, ис пользуя этот URL. В противном случае Buildroot предположит, что файл для загрузки находится на LIBFOO\_SITE. Buildroot не будет ничего делать с этими дополнительными файлами, кроме как загрузить их: их ис пользование из \$(LIBFOO\_DL\_DIR) будет зависеть от рецепта пакета.

- LIBFOO\_SITE\_METHOD определяет метод, ис используемый для извлечения или копирования исходного кода пакета. Во многих случаях Buildroot использует метод из содержимого LIBFOO\_SITE, и тройка LIBFOO\_SITE\_METHOD не нужна. Если HOST\_LIBFOO\_SITE\_METHOD не указан, по умолчанию ис пользуется значение LIBFOO\_SITE\_METHOD.

Возможные значения LIBFOO\_SITE\_METHOD:

- wget для обычных загрузок tarballs по FTP/HTTP. Используется по умолчанию, когда LIBFOO\_SITE начинается с http://, https:// или ftp://.
- scp для загрузки tarballs через SSH с scp. Используется по умолчанию, когда LIBFOO\_SITE начинается с scp://. - sftp для загрузки tarballs через SSH с sftp. Используется по умолчанию, когда LIBFOO\_SITE начинается с sftp://.

- svn для извлечения исходного кода из репозитория Subversion. Используется по умолчанию, когда LIBFOO\_SITE начинается с svn://. Если в LIBFOO\_SITE указан URL-адрес репозитория Subversion http://, необходимо указать LIBFOO\_SITE\_METHOD=svn.

Buildroot выполняет извлечение, которое сохраняется в виде tar-архива в кэше загрузок; последующие сборки используют tar-архив вместо выполнения еще одной извлечения.
  - cvs для извлечения исходного кода из репозитория CVS. Используется по умолчанию, когда LIBFOO\_SITE начинается с cvs://. Загруженный исходный код кэшируется как и в методе svn. Предполагается японский режим pserver, в противном случае он явно определен на LIBFOO\_SITE. Принимают такие как LIBFOO\_SITE=cvs://libfoo.net:/cvsroot/libfoo, так и LIBFOO\_SITE=cvs:///, в первом случае предполагается японский режим дос тупа pserver. LIBFOO\_SITE должен содержать исходный URL, а также удаленный каталог репозитория. Модуль – это имя пакета. LIBFOO\_VERSION является обязательным и должен быть тем же, веткой или датой (например, "2014-10-20", "2014-10-20 13:45", "2014-10-20 13:45+01" с мануалом "man cvs" для получения дополнительных сведений). – git для извлечения исходного кода из репозитория Git. Используется по умолчанию, когда LIBFOO\_SITE начинается с git://. Загруженный исходный код кэшируется как и в случае с методом svn.
  - hg для получения исходного кода из репозитория Mercurial. Необходимо указать LIBFOO\_SITE\_METHOD=hg, когда LIBFOO\_SITE содержит URL репозитория Mercurial. Загруженный исходный код кэшируется как и в случае с методом svn.
  - bzr для извлечения исходного кода из репозитория Bazaar. Используется по умолчанию, когда LIBFOO\_SITE начинается с bzr://. Загруженный исходный код кэшируется как и в случае с методом svn.
  - файл для локального отарбала. Следует использовать это, когда LIBFOO\_SITE указывает tarball пакета как локальное имя файла. Полезно для программ обес печения, которое не дистрибутируется или в системе контроля версий.
  - local для локального каталога исходного кода. Его следует использовать, когда LIBFOO\_SITE указывает путь к локальному каталогу, содержащему исходный код пакета. Buildroot копирует содержимое исходного каталога в каталог с боржами пакета. Обратите внимание, что для локальных пакетов никакие изменения не применяются. Если вам все равно нужно изменить исходный код, используйте LIBFOO\_POST\_RSYNC\_HOOKS, см. раздел 18.23.
- LIBFOO\_GIT\_SUBMODULES можно установить на YES, чтобы создать архив с подмодулями git в репозитории. Это дистрибутируется только для пакетов, загруженных с помощью git (т.е. когда LIBFOO\_SITE\_METHOD=git). Обратите внимание, что мы стараемся использовать такие подмодули git, если они содержат вложенные библиотеки, в этом случае мы предпочитаем использовать эти библиотеки из их собственного пакета.
  - LIBFOO\_GIT\_LFS следует установить на YES, если репозиторий Git использует Git LFS для хранения больших файлов вне диапазона. Это только дистрибутируется для пакетов, загруженных с помощью git (т.е. когда LIBFOO\_SITE\_METHOD=git).
  - LIBFOO SVN\_EXTERNALS можно установить в YES для создания архива с внешними ссылками svn. Это дистрибутируется только для пакетов, загруженных с помощью subversion.
  - LIBFOO\_STRIP\_COMPONENTS – это количество ведущих компонентов (каталогов), которые tar должен удалить из имен файлов при извлечении. В tar-архиве большинства пакетов есть один ведущий компонент с именем "<pkg-name>-<pkg-version>", поэтому Buildroot передает --strip-components=1 в tar, чтобы удалить его. Для нестандартных пакетов, в которых нет этого компонента или в которых есть более одного ведущего компонента для удаления установите эту переменную со значением, которое будет передано в tar. По умолчанию: 1.
  - LIBFOO\_EXCLUDES – это разделенный пробелами список шаблонов, которые следует исключить при извлечении архива. Каждый элемент из этого списка передается как опция tar --exclude. По умолчанию пусто.
  - LIBFOO\_DEPENDENCIES перечисляет зависимости (в терминах имени пакета), которые требуются для компиляции текущего целевого пакета. Эти зависимости гарантированно будут скомпилированы и установлены до начала настройки текущего пакета. Однако изменения в настройке этих зависимостей не приведут к пересборке текущего пакета. Аналогичным образом, HOST\_LIBFOO\_DEPENDENCIES перечисляет зависимости для текущего пакетах остава. Это используется только внутри инфраструктуры пакета и обычно не должно использоваться пакетами напрямую.
  - LIBFOO\_EXTRACT\_DEPENDENCIES перечисляет зависимости (в терминах имени пакета), которые требуются для извлечения текущего целевого пакета. Эти зависимости гарантированно будут извлечены и скомпилированы (но не обязательно построены) до извлечения текущего пакета. Аналогичным образом, HOST\_LIBFOO\_PATCH\_DEPENDENCIES перечисляет зависимости для текущего пакетах остава. Это используется редко, обычно LIBFOO\_DEPENDENCIES – это то, что вам действительно нужно.
  - LIBFOO\_PROVIDES перечисляет все виртуальные пакеты, реализации которых являются яlibfoo. См. раздел 18.12.

- LIBFOO\_INSTALL\_STAGING может быть установлено на YES или NO (по умолчанию). Если установлено на YES, то команды в LIBFOO\_INSTALL\_STAGING переменные выполняются для установки пакета в промежуточный каталог.
- LIBFOO\_INSTALL\_TARGET может быть установлен на YES (по умолчанию) или NO. Если установлено на YES, то команды в LIBFOO\_INSTALL\_TARGET переменные выполняются для установки пакета в целевой каталог.
- LIBFOO\_INSTALL\_IMAGES может быть установлено на YES или NO (по умолчанию). Если установлено на YES, то команды в LIBFOO\_INSTALL\_IMAGES переменные выполняются для установки пакета в каталог образов.
- LIBFOO\_CONFIG\_SCRIPTS перечисляет имена файлов в \$(STAGING\_DIR)/usr/bin, которые требуют специального исключения, чтобы сделять их дружественными крос-компиляции. Можно указать несколько имен файлов, разделенных пробелом, и все они относятся к \$(STAGING\_DIR)/usr/bin. Файлы, перечисленные в LIBFOO\_CONFIG\_SCRIPTS, также удаляются из \$(TARGET\_DIR)/usr/bin, поскольку они не нужны на целевой системе.
- LIBFOO\_DEVICES перечисляет файлы устройств, которые будут созданы Buildroot при использовании статической таблицы устройств. Синтаксис для использования makedevs one. Вы можете найти документацию по этому синтаксису в Главе 25. Эта переменная не обязательна.
- LIBFOO\_PERMISSIONS перечисляет изменения разрешений, которые необходимо сделать в конце процесса сборки. Синтаксис с новой makedevs one. Вы можете найти документацию по этому синтаксису в Главе 25. Эта переменная не обязательна.
- LIBFOO\_USERS перечисляет пользователей, которых нужно создать для этого пакета, если он устанавливается в программу, которую вы хотите запустить как определенный пользователь (например, как демон или как cron-job). Синтаксис подходит для makedevs и описан в Главе 26. Эта переменная не обязательна.
- LIBFOO\_LICENSE определяет лицензию (или лицензии), под которой выпущен пакет. Это имя будет отображаться в файле манифеста, созданного make legal-info. Если лицензия отображается в списке лицензий SPDX, используйте короткий идентификатор SPDX, чтобы сделать файл манифеста однородным. В противном случае опишите лицензию точно и кратко, избегая двусмысленных имен, таких как BSD, которые на самом деле называют семейством лицензий. Эта переменная не обязательна. Если она не определена, в поле лицензии файла манифеста для этого пакета появится ярлык unknown.

Ожидаемый формат этой переменной должен соответствовать следующим правилам:

- Если разные части пакета выпущены под разными лицензиями, то разделите лицензии запятыми (например, LIBFOO\_LICENSE = GPL-2.0+, LGPL-2.1+). Если есть четкое различие между тем, какой компонент лиценизируется под какой лицензией, то аннотируйте лицензию с этим компонентом в скобках (например, LIBFOO\_LICENSE = GPL-2.0+ (программы), LGPL-2.1+ (библиотеки)).
- Если некоторые лицензии зависят от включения подопечных, добавьте условные лицензии через запятую (например: GPL-2.0+ (программы); FOO\_LICENSE += , инфраструктура автоматически удалит пробел перед запятой).
- Если пакет имеет двойную лицензию, то разделите лицензии с помощью ключевого слова или (например, LIBFOO\_LICENSE = AFL-2.1 или GPL-2.0+).
- LIBFOO\_LICENSE\_FILES — это разделенный пробелами список файлов tarball пакета, которые содержат лицензию (и), под которой выпущен пакет. make legal-info копирует все эти файлы в каталог legal-info. Для получения дополнительной информации см. Главу 13. Эта переменная не обязательна. Если она не определена, будет выдано предупреждение, чтобы сообщить вам об этом, и в поле файлов лицензий файла манифеста для этого пакета появится сообщение not saved.
- LIBFOO\_ACTUAL\_SOURCE\_TARBALL применяется только к пакетам, чьи параметры LIBFOO\_SITE / LIBFOO\_SOURCE указывают на архив, который на самом деле содержит не исходный код, а двоичный код. Это очень редкий случай, известный только для внешних наборов инструментов, которые состоят из уже скомпилированных, хотя теоретически он может применяться и к другим пакетам. В таких случаях отдельный tarball обычно доступен с фактическим исходным кодом. Установите LIBFOO\_ACTUAL\_SOURCE\_TARBALL на имя фактического архива исходного кода, и Buildroot загрузит его и будет использовать при запуске make legal-info для сборки реальной и значимой материала. Обратите внимание, что этот файл не будет загружен ни во время обычных сборок, ни при make source.
- LIBFOO\_ACTUAL\_SOURCE\_SITE предсталяет местоположение фактического исходного tarball. Значение по умолчанию — LIBFOO\_SITE, поэтому Вам не нужно устанавливать эту переменную, если двоичные исходные архивы размещены в одном каталоге. Если LIBFOO\_ACTUAL\_SOURCE\_ не установлена, то нет способа определять LIBFOO\_ACTUAL\_SOURCE\_SITE.
- LIBFOO\_REDISTRIBUTE может быть установлено на YES (по умолчанию) или NO, чтобы указать, разрешено ли распространять исходный код пакета. Установите его на NO для пакетов без открытого исходного кода: Buildroot не будет сортировать исходный код для этого пакета при сборке реальной информации.

- LIBFOO\_FLAT\_STACKSIZE определяет размер стека приложения в троичном формате FLAT. Размер стека приложения на процессорах с архитектурой NOMMU не может быть увеличен во время выполнения. Размер стека по умолчанию для двоичного формата FLAT составляет всего 4 Кбайт. Если приложение потребляет больше места, добавьте здесь требуемое число.
- LIBFOO\_BIN\_ARCH\_EXCLUDE — это разделенный пробелами список путей (относительно корневого каталога), которые следуют идентификатором проверки того, что пакет устанавливает правильно кросс-компилированные двоичные файлы. Вам редко нужно устанавливать эту переменную, если только пакет не устанавливает двоичные файлы вне местоположений по умолчанию, /lib/firmware, /usr/lib/firmware, /lib/modules, /usr/lib/modules и /usr/share, которые автоматически исключаются.
- LIBFOO\_IGNORE CVES — это разделенный пробелами список CVE, который сообщает инструментам отслеживания CVE Buildroot, какие CVE следуют идентификатором проверки для этого пакета. Обычно это используется, когда CVE исправлено патчем в пакете или когда CVE по какой-то причине не влияет на пакет Buildroot. Комментарий Makefile всегда должен предшествовать добавлению CVE в эту переменную.

Пример:

```
# 0001-fix-cve-2020-12345.patch LIBFOO_IGNORE CVEs
+= CVE-2020-12345 # только при сборке с помощью libbaz,
который Buildroot не поддерживает LIBFOO_IGNORE CVEs += CVE-2020-54321
```

- Переменные LIBFOO\_CPE\_ID\_\* — это набор переменных, позволяющих пакету определять свой идентификатор CPE. Доступные переменные:

- LIBFOO\_CPE\_ID\_VALID, если установлено значение YES, указывает, что значения по умолчанию для каждой из следующих переменных являются яподходящими, и генерирует действительный идентификатор CPE.
- LIBFOO\_CPE\_ID\_PREFIX, указывает префикс идентификатора CPE, т.е. первые три поля Если не определено, значение по умолчанию — cpe:2.3:a.
- LIBFOO\_CPE\_ID\_VENDOR, указывает часть поставщика идентификатора CPE. Если не определено, значение по умолчанию равно <имя\_пакета>\_проект.
- LIBFOO\_CPE\_ID\_PRODUCT, указывает часть продукта идентификатора CPE. Если не определено, значение по умолчанию равно <имя\_пакета>.
- LIBFOO\_CPE\_ID\_VERSION, указывает версию идентификатора CPE. Если не определено, значение по умолчанию равно \$(LIBFOO\_VERSION).
- LIBFOO\_CPE\_ID\_UPDATE указывает часть обновления идентификатора CPE. Если не определено, значение по умолчанию равно \*.

Если какая-либо из этих переменных определена, то инфраструктура общего пакета предполагает, что пакет предоставляет действительную информацию CPE. В этом случае инфраструктура общего пакета определит LIBFOO\_CPE\_ID.

Для этого пакета, если его переменные LIBFOO\_CPE\_ID\_\* не определены, он наследует значение этих переменных из соответствующего корневого пакета.

Для определения этих переменных рекомендуется использовать следующий синтаксис:

```
ВЕРСИЯ_ЛИБФУ = 2.32
```

Теперь переменные, которые определяют, что должно выполняться на различных этапах процесса сборки.

- LIBFOO\_EXTRACT\_CMDS перечисляет действия, которые необходимо выполнить для извлечения пакета. Обычно это не требуется, поскольку tarballы автоматически обрабатываются Buildroot. Однако, если пакет использует нестандартный формат архива, например файл ZIP или RAR, или имеет tarball с нестандартной организацией, эта переменная позволяет определить поведение инфраструктуры пакета по умолчанию.
- LIBFOO\_CONFIGURE\_CMDS перечисляет действия, которые необходимо выполнить для настройки пакета перед его компиляцией.
- LIBFOO\_BUILD\_CMDS перечисляет действия, которые необходимо выполнить для компиляции пакета.
- HOST\_LIBFOO\_INSTALL\_CMDS перечисляет действия, которые необходимо выполнить для установки пакета, когда пакет является ядром системы. Пакет должен установить свои файлы в каталог, указанный \$(HOST\_DIR). Все файлы, включая файлы разработки, такие как заголовки, должны быть установлены, поскольку другие пакеты могут быть скомпилированы поверх этого пакета.

- LIBFOO\_INSTALL\_TARGET\_CMDS перечисляет действия, которые необъодимо выполнить для установки пакета в целевой каталог, когда пакет является целевым пакетом. Пакет должен устанавливаться с его файлами в каталог, указанный \$(TARGET\_DIR). Должны быть установлены только файлы, необъодимые для выполнения пакета. Заголовочные файлы, статические библиотеки и документация будут с нова удалены, когда целевая файловая система будет завершена.
- LIBFOO\_INSTALL\_STAGING\_CMDS перечисляет действия, которые необъодимо выполнить для установки пакета в промежуточный каталог, когда пакет является целевым пакетом. Пакет должен устанавливаться с его файлами в каталог, указанный \$(STAGING\_DIR). Всё файлы разработки должны быть установлены, так как они могут потребоваться для компиляции других пакетов.
- LIBFOO\_INSTALL\_IMAGES\_CMDS перечисляет действия, которые необъодимо выполнить для установки пакета в каталог images, когда пакет является целевым пакетом. Пакет должен устанавливаться с его файлами в каталог, заданный \$(BINARIES\_DIR). Здесь следует размещать только файлы, которые являются двоичными образами (т. е. образами), которые не принадлежат TARGET\_DIR, но необъодимы для загрузки и платы. Например, пакет должен использовать этот шаг, если у него есть двоичные файлы, которые будут положены на образ ядра, загрузчик или образы корневой файловой системы.
- LIBFOO\_INSTALL\_INIT\_SYSV, LIBFOO\_INSTALL\_INIT\_OPENRC и LIBFOO\_INSTALL\_INIT\_SYSTEMD перечисляют действия по установке скриптов инициализации для систем инициализации типа systemV (busybox, sysvinit и т. д.), openrc или для модулей systemd. Эти команды будут выполняться только при установке соответствующей системы инициализации (т. е. если в конфигурации включено включение системы инициализации выбран systemd, будет выполняться только LIBFOO\_INSTALL\_INIT\_SYSTEMD). Если установленный ключением является случайным, когда включено включение системы инициализации выбран openrc, а LIBFOO\_INSTALL\_INIT\_OPENRC не установлен, в такой ситуации будет вызываться LIBFOO\_INSTALL\_INIT\_SYSV, поскольку openrc поддерживает скрипты инициализации sysv. Когда systemd используется как система инициализации, buildroot автоматически включает все службы с помощью команды systemctl preset-all на заключительном этапе построения образа. Вы можете добавить файлы предустановки, чтобы предотвратить автоматическое включение определенного блока buildroot.
- LIBFOO\_HELP\_CMDS перечисляет действия для печати с правкой по пакету, которая включена в основной вывод с правкой make. Эти команды могут печатать что угодно в любом формате. Это используется редко, так как пакеты редко имеют пользовательские правила. Не используйте эту переменную, если вы действительно не знаете, что вам нужно распечатать с правкой.
- LIBFOO\_LINUX\_CONFIG\_FIXUPS перечисляет параметры конфигурации ядра Linux, которые необъодимы для сборки и использования этого пакета, и без которых пакет в корне сломан. Это должен быть набор вызовов одного из параметров настройки kconfig: KCONFIG\_ENABLE\_OPT, KCONFIG\_DISABLE\_OPT или KCONFIG\_SET\_OPT. Это используется редко, так как пакет обычно не имеет строгих требований к параметрам ядра.

Предпочтительный способ определения этих переменных:

```
определить LIBFOO_CONFIGURE_CMDS
    действие 1
    действие 2
    действие 3
отпустить
```

В определениях действий вы можете использовать следующие переменные:

- \$(LIBFOO\_PKGDIR) содержит путь к каталогу, содержащему файлы libfoo.mk и Config.in. Эта переменная полезна, когда необходимо установить файл, упакованный в Buildroot, например файл конфигурации времени выполнения изображение заголовка...
- \$(@D), содержащий каталог, в котором был распакован исходный код пакета.
- \$(LIBFOO\_DL\_DIR) содержит путь к каталогу, в котором хранятся загрузчики, сделанные Buildroot для libfoo.
- \$(TARGET\_CC), \$(TARGET\_LD) и т. д. для получения целевых утилит компиляции
- \$(TARGET\_CROSS) для получения префикса архитектурных инструментов компиляции
- Конечно, переменные \$(HOST\_DIR), \$(STAGING\_DIR) и \$(TARGET\_DIR) для правильной установки пакетов. Эти переменные указывают на глобальные каталоги хоста, промежуточного размещения и целевого размещения, если только не используется поддержка каталогов по пакетам, в этом случае они указывают на текущий каталог хоста, промежуточного размещения и целевого размещения пакета. В обоих случаях это не имеет никакого значения с точки зрения пакета он должен просто использовать HOST\_DIR, STAGING\_DIR и TARGET\_DIR. Подробнее о поддержке каталогов по пакетам см. в разделе 8.12.

Наконец, вы также можете использовать хаки. Подробнее см. в разделе 18.23.

## 18.7 Инфраструктура для пакетов на основе autotools

### 18.7.1 Учебное пособие по пакету autotools

Сначала давайте посмотрим, как написать файл .mk для пакета на основе autotools, например:

```
01: #####  
02: #  
03: # libfoo  
04: #  
05: #####  
06:  
07: LIBFOO_VERSION = 1.0.08:  
LIBFOO_SOURCE = libfoo-$LIBFOO_VERSION.tar.gz 09: LIBFOO_SITE = http://  
www.foosoftware.org/download 10: LIBFOO_INSTALL_STAGING = DA  
  
11: LIBFOO_INSTALL_TARGET = YES  
12: LIBFOO_CONF_OPTS = --disable-shared  
13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf 14:  
  
15: $(eval $(package-autotools))
```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tar-архива (рекомендуется xz-ed tar-архив) и место расположение tar-архива в Интернете. Buildroot автоматически загрузит tarball из этого места.

В строке 10 мы говорим Buildroot установить пакет в промежуточный каталог .промежуточный каталог , расположенный в output/staging/ , — это каталог , в котором установлены все пакеты, включая файлы разработки и т. д. По умолчанию пакеты не устанавливаются в промежуточный каталог , поскольку обычно в промежуточный каталог нужно устанавливать только библиотеки: их файлы разработки нужны для компиляции других библиотек или приложений, зависящих от них . Также по умолчанию, когда включена промежуточная установка, пакеты устанавливаются в этом месте с помощью команды make install.

В строке 11 мы говорим Buildroot не устанавливать пакет в целевой каталог . Этот каталог содержит корневой файловой системой, запущенной нацели. Для числа статических библиотек нет необходимости одновременно устанавливать их в целевой каталог , поскольку они не будут использоваться вовремя выполнения. По умолчанию целевая установка включена; установка этой переменной в значение NO почти никогда не требуется. Также по умолчанию пакеты устанавливаются в этом месте с помощью команды make install.

В строке 12 мы сообщаем Buildroot о необходимости передать пользовательский параметр конфигурации, который будет передан скрипту ./configure перед настройкой и сборкой пакета.

В строке 13 мы объявляем наши зависимости, чтобы они были построены до начала процесса сборки нашего пакета.

Наконец, в строке 15 мы вызываем макрос autotools-package , который генерирует все правила Makefile , которые фактически позволяют обратиться к пакету.

### 18.7.2 Справочник по пакетам autotools

Основной макрос инфраструктуры пакета autotools — autotools-package . Он основан на макросе generic-package . Возможно иметь целевые и хостовые пакеты также доступными с помощью макроса host-autotools-package .

Как и общая инфраструктура, инфраструктура autotools работает путем определения ряда переменных перед вызовом макроса autotools-package .

Все переменные метаданных пакета, которые используются в общей инфраструктуре пакета, также используются в инфраструктуре autotools .

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры autotools . Многие из них полезны только в очень специфических случаях, поэтому типичные пакеты будут использовать только некоторые из них .

- LIBFOO\_SUBDIR может содержать имя подкаталога внутри пакета, содержащего скрипты конфигурации. Это полезно, если, например, основной скрипт конфигурации не находится в корне дерева, извлеченного из tarball . Если HOST\_LIBFOO\_SUBDIR не указан, по умолчанию используется LIBFOO\_SUBDIR .

- LIBFOO\_CONF\_ENV, чтобы указать дополнительные переменные с реды для передачи в скрипты конфигурации. По умолчанию пусто.
- LIBFOO\_CONF\_OPTS, чтобы указать дополнительные параметры конфигурации для передачи в скрипты конфигурации. По умолчанию пусто.
- LIBFOO\_MAKE, чтобы указать альтернативную команду make. Обычно это полезно, когда в конфигурации включен параллельный make (с помощью BR2\_JLEVEL), но эта функция должна быть отключена для данного пакета по той или иной причине. По умолчанию установлено значение \$(MAKE). Если пакет не поддерживает параллельную сборку, то следует установить значение LIBFOO\_MAKE=\$(MAKE1).
- LIBFOO\_MAKE\_ENV, чтобы указать дополнительные переменные с реды для передачи в make на этапе сборки. Они передаются перед командой make. По умолчанию пусто.
- LIBFOO\_MAKE\_OPTS, чтобы указать дополнительные переменные для передачи в make на этапе сборки. Они передаются после команды make. По умолчанию пусто.
- LIBFOO\_AUTORECONF, сообщает, следует ли автоматически перенасыщать пакет или нет (т.е. следует ли повторно генерировать скрипты конфигурации и файлы Makefile.in путем повторного запуска autoconf, automake, libtool и т.д.). Допустимые значения YES и NO. По умолчанию значение равно NO.
- LIBFOO\_AUTORECONF\_ENV, чтобы указать дополнительные переменные с реды для передачи в программу autoreconf, если LIBFOO\_AUTORECONF. Они передаются с редом команды autoreconf. По умолчанию пусто.
- LIBFOO\_AUTORECONF\_OPTS для указания дополнительных параметров, передаваемых программе autoreconf, если LIBFOO\_AUTORECONF=YES. По умолчанию пусто.
- LIBFOO\_AUTOPOINT, сообщает, следует ли автоматически указывать пакет или нет (т.е. нужны ли пакету инфраструктура I18N. Скопировано). Действительно только при LIBFOO\_AUTORECONF=YES. Допустимые значения YES и NO. Значение по умолчанию: NO.
- LIBFOO\_LIBTOOL\_PATCH сообщает, следует ли применять исправление Buildroot для исправления проблем кросскомпиляции libtool. Допустимые значения DA и NE. По умолчанию значение — DA.
- LIBFOO\_INSTALL\_STAGING\_OPTS содержит параметры make, используемые для установки пакета в промежуточный каталог. По умолчанию значение равно DESTDIR=\$(STAGING\_DIR) install, что является правильным для большинства пакетов autotools. Его также можно переопределить.
- LIBFOO\_INSTALL\_TARGET\_OPTS содержит параметры make, используемые для установки пакета в целевой каталог. По умолчанию значение равно DESTDIR=\$(TARGET\_DIR) install. Значение по умолчанию является правильным для большинства пакетов autotools, но его также можно переопределить при необходимости.

С инфраструктурой autotools все шаги, необязательные для сборки и установки пакетов, уже определены, и они, как правило, хорошо работают для большинства пакетов на основе autotools. Однако, при необходимости, всегда возможно настроить то, что делается на каждом конкретном шаге:

- Добавляя постоперационный шаг (после извлечения, исправления, настройки, сборки или установки). Подробности см. в разделе [18.23](#).
- Переопределяя один из шагов. Например, даже если используется инфраструктура autotools, если файл пакета .mk определяет собственную переменную LIBFOO\_CONFIGURE\_CMDS, она будет использоваться вместо переменной autotools по умолчанию. Однако использование этого метода должно быть ограничено очень конкретными случаями. Не используйте его в общем случае.

## 18.8 Инфраструктура для пакетов на основе CMake

### 18.8.1 Учебник по пакету cmake

Сначала давайте посмотрим, как написать файл .mk для пакета на основе CMake, на примере:

01: ## ...

06:

```

07: LIBFOO_VERSION = 1.0 08: LIBFOO_SOURCE
= libfoo-$(LIBFOO_VERSION).tar.gz 09: LIBFOO_SITE = http://www.foosoftware.org/download 10:
LIBFOO_INSTALL_STAGING =да 11: LIBFOO_INSTALL_TARGET = нет 12: LIBFOO_CONF_OPTS =
-DBUILD_DEMOS=ON

13: LIBFOO_DEPENDENCIES = libglib2 host-pkgconf 14:

15: $(eval $(cmake-package))

```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tar-архива (рекомендуется xz-ed tar-архив) и место расположение tar-архива в Интернете.

Buildroot автоматически разбирает tarball из этого места.

В строке 10 мы говорим Buildroot установить пакет в промежуточный каталог .промежуточный каталог, расположенный в output/staging/, — это каталог, в котором установлены все пакеты, включая их файлы разработки и т. д. По умолчанию пакеты не устанавливаются в промежуточный каталог, поскольку обычно в промежуточный каталог нужно устанавливать только библиотеки: их файлы разработки нужны для компиляции других библиотек или приложений, зависящих от них. Также по умолчанию, когда включена промежуточная установка, пакеты устанавливаются в этом месте с помощью команды make install.

В строке 11 мы говорим Buildroot не устанавливать пакет в центральный каталог .Этот каталог содержит то, что станет корневой файловой системой, запущенной нацели. Для числа статических библиотек нет необходимости устанавливать их в центральный каталог, поскольку они не будут использоваться во время выполнения. По умолчанию центральная установка включена в установке этой переменной в значение NO почти никогда не требуется. Также по умолчанию пакеты устанавливаются в этом месте с помощью команды make install.

В строке 12 мы сообщаем Buildroot о необходимости передавать пользовательские параметры в CMake при настройке пакета.

В строке 13 мы объявляем наши зависимости, чтобы они были прошиты до начала процесса сборки нашего пакета.

Наконец, в строке 15 мы вызываем макрос cmake-package, который генерирует все правила Makefile, которые фактически позволяют обратиться к пакету.

## 18.8.2 Ссылка на пакет cmake

Основной макрос инфраструктуры пакетов CMake — cmake-package. Он основан на макросе generic-package.

Возможность иметь центральные и хостовые пакеты также доступна с помощью макроса host-cmake-package.

Как и большинство инфраструктур, инфраструктура CMake работает путем определения ряда переменных перед вызовом cmake-packa. макроса.

Все переменные метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре CMake.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры CMake. Многие из них полезны только очень специфических случаях, поэтому типичные пакеты будут использовать только некоторые из них.

- LIBFOO\_SUBDIR может содержать имя подкаталога внутри пакета, содержащего основной файл CMakeLists.txt. Это полезно, если, например, основной файл CMakeLists.txt не находится в корне дерева, извлеченного tarball. Если HOST\_LIBFOO\_SUBDIR не указан, по умолчанию используется LIBFOO\_SUBDIR.
- LIBFOO\_CMAKE\_BACKEND указывает на используемый бэкэнд cmake, один из make (для использования генератора GNU Makefiles, по умолчанию) или ninja (для использования генератора Ninja).
- LIBFOO\_CONF\_ENV, чтобы указать дополнительные переменные передачи в CMake. По умолчанию пусто. Ряд общих параметров CMake задается инфраструктурой cmake-package; поэтому обычно нет необходимости задавать их в файле \*.mk пакета, если только вы не хотите их переопределить:
  - CMAKE\_BUILD\_TYPE управляет BR2\_ENABLE\_RUNTIME\_DEBUG;
  - CMAKE\_INSTALL\_PREFIX;
- LIBFOO\_CONF\_OPTS, чтобы указать дополнительные параметры конфигурации для передачи в CMake. По умолчанию пусто. Ряд общих параметров CMake задается инфраструктурой cmake-package; поэтому обычно нет необходимости задавать их в файле \*.mk пакета, если только вы не хотите их переопределить:

- BUILD\_SHARED\_LIBS управляет BR2\_STATIC\_LIBS;
  - BUILD\_DOC, BUILD\_DOCS отключены;
  - BUILD\_EXAMPLE, BUILD\_EXAMPLES отключены;
  - BUILD\_TEST, BUILD\_TESTS, BUILD\_TESTING отключены.
- LIBFOO\_BUILD\_ENV и LIBFOO\_BUILD\_OPTS для указания дополнительных переменных с реды или параметров командной строки, для передачи бэкэнду во время сборки.
  - LIBFOO\_SUPPORTS\_IN\_SOURCE\_BUILD = NO следует устанавливать, если пакет не может быть собран внутри исходного дерева, но требует отдельного каталога с сборки.
  - LIBFOO\_MAKE, чтобы указать альтернативную команду make. Обычно это полезно, когда в конфигурации включен параллельный make (с помощью BR2\_JLEVEL), но эта функция должна быть отключена для данного пакета по той или иной причине. По умолчанию установлено значение \$(MAKE). Если пакет не поддерживает параллельную сборку, то следует установить значение LIBFOO\_MAKE=\$(MAKE1).
  - LIBFOO\_MAKE\_ENV, чтобы указать дополнительные переменные с реды для передачи в make на этапе с сборки. Они передаются перед командой make. По умолчанию пусто.
  - LIBFOO\_MAKE\_OPTS, чтобы указать дополнительные переменные для передачи в make на этапе с сборки. Они передаются после команды make. По умолчанию пусто.
  - LIBFOO\_INSTALL\_OPTS содержит параметры make, используемые для установки пакета в каталог хоста. По умолчанию значение is install, что верно для большинства пакетов CMake. Его также можно переопределить.
  - LIBFOO\_INSTALL\_STAGING\_OPTS содержит параметры make, используемые для установки пакета в промежуточный каталог. По умолчанию значение равно DESTDIR=\$(STAGING\_DIR) install/fast, что является правильным для большинства пакетов CMake. Его также можно переопределить.
  - LIBFOO\_INSTALL\_TARGET\_OPTS содержит параметры make, используемые для установки пакета в целевой каталог. По умолчанию значение равно DESTDIR=\$(TARGET\_DIR) install/fast. Значение по умолчанию является правильным для большинства пакетов CMake, но его также можно переопределить при необходимости.

С инфраструктурой CMake все шаги, необходимые для сборки и установки пакетов, уже определены, и они, как правило, хорошо работают для большинства пакетов на основе CMake. Однако, при необходимости, все еще возможно настроить то, что делается на каждом конкретном шаге:

- Добавляя пост-операционный хук (после извлечения, исправления, настройки, сборки или установки). Подробности см. в разделе [18.23](#).
- Переопределяя один из шагов. Например, даже если используется инфраструктура CMake, если файл пакета .mk определяет собственную переменную LIBFOO\_CONFIGURE\_CMDS, она будет использоваться вместе с переменной CMake по умолчанию. Однако использование этого метода должно быть ограничено очень конкретными случаями. Не используйте его в общем случае.

## 18.9 Инфраструктура для пакетов Python

Эта инфраструктура применяется к пакетам Python, которые используют стандартные механизмы Python setuptools, pep517, flit или maturin в качестве своей системы сборки, обычно узнаваемой по использованию скрипта setup.py или файла pyproject.toml.

### 18.9.1 Учебник по пакету python

Сначала давайте посмотрим, как написать файл .mk для пакета Python, на примере:

```
01: ## ...
```

```
06:
```

```

07: PYTHON_FOO_VERSION = 1.0 08: PYTHON_FOO_SOURCE
= python-foo-${PYTHON_FOO_VERSION}.tar.xz 09: PYTHON_FOO_SITE = http://www.foosoftware.org/download 10:
PYTHON_FOO_LICENSE = BSD-3-Clause 11: PYTHON_FOO_LICENSE_FILES = Лицензия 12: PYTHON_FOO_ENV =
НЕ КОТОРАЯ_ПЕРЕМЕННАЯ=1

13: PYTHON_FOO_DEPENDENCIES = libmad
14: PYTHON_FOO_SETUP_TYPE = setuptools 15:

16: $(eval $(python-package))

```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tar-архива (рекомендуется xxz-ed tar-архив) и место расположение tar-архива в Интернете.

Buildroot автоматически загружает tarball из этого места.

В строках 10 и 11 мы приводим лицензионные данные пакета (его лицензию в строке 10 и файл, содержащий текст лицензии, в строке 11).

В строке 12 мы сообщаем Buildroot о необходимости передавать пользовательские параметры скрипту Python setup.py при настройке пакета.

В строке 13 мы объявляем наши зависимости, чтобы они были построены до начала процесса сборки нашего пакета.

В строке 14 мы объявляем конкретную систему с борками Python. В этом случае используется система с борками Python setuptools.

Поддерживаются семь из них: flit, hatch, pep517, poetry, setuptools, setuptools-rust и maturin.

Наконец, в строке 16 мы вызываем макрос python-package, который генерирует все правила Makefile, которые фактически позволяют собрать пакет.

## 18.9.2 Справочник по пакетам python

В качестве политики, пакеты, которые просто предстаивают модули Python, должны все называться `python-<something>` в Buildroot. Другие пакеты, которые используют систему с борками Python, но не являются модулями Python, могут свободно выбирать свое имя (существующие примеры в Buildroot — `scons` и `supervisor`).

Основной макрос инфраструктуры пакетов Python — `python-package`. Он похож на макрос `generic-package`.

Также можно создавать пакеты хоста Python с помощью макроса `host-python-package`.

Как и общая инфраструктура, инфраструктура Python работает путем определения ряда переменных перед вызовом макросов `python-package` или `host-python-package`.

Все переменные метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре Python.

Обратите внимание, что:

- Нет необходимости добавлять `python` или `host-python` в переменную `PYTHON_FOO_DEPENDENCIES` пакета, поскольку эти базовые зависимости автоматически добавляются по мере необходимости инфраструктурой пакетов Python.
- Аналогично, не нужно добавлять `host-python-setuptools` в `PYTHON_FOO_DEPENDENCIES` для основанных на `setuptools` пакетов, поскольку он автоматически добавляется инфраструктурой Python по мере необходимости.

Одна переменная специфичная для инфраструктуры Python, является обязательной:

- `PYTHON_FOO_SETUP_TYPE`, чтобы определить, какая система с борками Python используется пакетом. Семь поддерживаемых значений: `flit`, `hatch`, `pep517`, `poetry`, `setuptools`, `setuptools-rust` и `maturin`. Если вы не знаете, какая из них используется в вашем пакете, посмотрите на файл `setup.py` или `pyproject.toml` в исходном коде вашего пакета и посмотрите, импортирует ли он что-то из модуля `flit` или модуля `setuptools`. Если пакет использует файл `pyproject.toml` без каких-либо требований к системе с борками и с локальным внутренним путем к `backend-path`, следуйте его использовать `pep517`.

В зависимости от потребностей пакета можно опционально определить несколько дополнительных переменных, специфичных для инфраструктуры Python.

Многие из них полезны только в очень специфических случаях, поэтому типичные пакеты будут использовать только некоторые из них или не будут использовать ни одного.

- PYTHON\_FOO\_SUBDIR может содержать имя подкаталога внутри пакета, содержащего новый файл setup.py или pyproject.toml. Это полезно, если, например, новый файл setup.py или pyproject.toml не находится в корне дерева, извлеченном из tarball. Если HOST\_PYTHON\_FOO\_SUBDIR не указан, по умолчанию используется PYTHON\_FOO\_SUBDIR.
- PYTHON\_FOO\_ENV, чтобы указать дополнительные переменные среды для передачи в скрипты Python setup.py (для пакетов setuptools) или в скрипты support/scripts/pyinstaller.py (для пакетов flit/pep517) для этапов сборки и установки. Обратите внимание, что инфраструктура автоматически передает несколько стандартных переменных, определенных в PKG\_PYTHON\_SETUP\_TOOLS\_ENV (для целевых пакетов setuptools), HOST\_PKG\_PYTHON\_SETUP\_TOOLS\_ENV (для пакетов setuptools), PKG\_PYTHON\_PEP51 (для целевых пакетов flit/pep517) и HOST\_PKG\_PYTHON\_PEP517\_ENV (для пакетов flit/pep517).
- PYTHON\_FOO\_BUILD\_OPTS, чтобы указать дополнительные параметры для передачи в скрипты Python setup.py на этапе сборки. Обычно это имеет смысл использовать только для пакетов на основе setuptools, поскольку пакеты на основе flit/pep517 не передают эти параметры в скрипты setup.py, а вместе с тем они передают их в support/scripts/pyinstaller.py.
- PYTHON\_FOO\_INSTALL\_TARGET\_OPTS, PYTHON\_FOO\_INSTALL\_STAGING\_OPTS, HOST\_PYTHON\_FOO\_INSTALL\_OP для указания дополнительных параметров для передачи в скрипты Python setup.py (для пакетов setuptools) или support/scripts/pyinstall (для пакетов flit/pep517) на этапе целевой установки, этапе промежуточной установки или установке хоста соответственно.

С инфраструктурой Python все шаги, необходимые для сборки и установки пакетов, уже определены, и они, как правило, хорошо работают для большинства пакетов на основе Python. Однако, при необходимости, все еще возможно настроить то, что делается на каждом конкретном шаге:

- Добавляя постоперационный хук (после извлечения исключения настройки, сборки или установки). Подробности см. в разделе [18.23](#).
- Переопределяя один из шагов. Например, даже если используется инфраструктура Python, если файл пакета .mk определяет собственную переменную PYTHON\_FOO\_BUILD\_CMDS, она будет использоваться вместе с переменной Python по умолчанию. Однако использование этого метода должно быть ограничено очень конкретными случаями. Не используйте его в общем случае.

### 18.9.3 Создание пакета python из репозитория PyPI

Если пакет Python, для которого вы хотите создать пакет Buildroot, доступен в PyPI, вы можете использовать инструмент scanpypi, расположенный в utils/, чтобы автоматизировать процесс.

Список существующих пакетов PyPI можно найти [здесь](#).

Для работы scanpypi необходимо, чтобы на вашем хосте был установлен пакет Python setuptools.

Нажмите в корневом каталоге buildroot, чтобы выполнить:

```
пакет utils/scanpypi foo bar -o
```

Это создаст пакеты python-foo и python-bar в папке пакетов, если они существуют на <https://pypi.python.org>.

Найдите меню внешних модулей python и вставьте туда свой пакет. Помните, что элементы внутри меню должны быть в алфавитном порядке.

Пожалуйста, имейте в виду, что вам, скорее всего, придется явно проверять пакет на наличие ошибок, поскольку есть вещи, которые генератор не может угадать (например, зависит от любого из новых модулей Python, таких как BR2\_PACKAGE\_PYTHON\_ZLIB).

Также, пожалуйста, примите во внимание, что лицензии файлы лицензии предполагают, что должны быть проверены. Вам также необходимо вручную добавить пакет в файл package/Config.in.

Если ваш пакет Buildroot находится в официальном дереве Buildroot, а в дереве br2-external, используйте флаг -o следующим образом:

```
utils/scanpypi foo bar -o other_package_dir
```

Это приведет к созданию пакетов python-foo и python-bar в other\_package\_directory вместо package.

Опция -h выведет список доступных опций:

```
утилиты/scanpypi -h
```

## 18.9.4 Серверная часть CFFI пакета python

Интерфейс внешних функций С для Python (CFFI) обеспечивает удобный и надежный способ вызова с компилированного кода С из Python с использованием докладчиков интерфейсов, написанных на языке С. Пакеты Python, использующие этот бэкэнд, можно определить по появлению зависимостей cffi в поле install\_requires их файла setup.py.

Такой пакет должен:

- добавить python-cffi как зависимость времени выполнения, чтобы установить с компилиированной оберткой библиотеки С на целевой платформе. Это достигается путем добавления select BR2\_PACKAGE\_PYTHON\_CFFI в пакет Config.in.

```
конфигурация BR2_PACKAGE_PYTHON_FOO
bool "python-foo" select
BR2_PACKAGE_PYTHON_CFFI # время выполнения
```

- добавить host-python-cffi как зависимость времени сборки для кросскомпиляции оболочки С. Это достигается путем добавления host-python-cffi в переменную PYTHON\_FOO\_DEPENDENCIES.

```
#####
#
# питон-foo #
#####
...
PYTHON_FOO_DEPENDENCIES = x от python-cffi
$(eval $(python-package))
```

## 18.10 Инфраструктура для пакетов на основе LuaRocks

### 18.10.1 luarocks-пакет учебник

Сначала давайте посмотрим, как написать файл .mk для пакета на основе LuaRocks, на примере:

```
01: #####
02: #
03: #luaифу
04: #
05: #####
06:
07: LUA_FOO_VERSION = 1.0.2-1
08: LUA_FOO_NAME_UPSTREAM = foo
09: LUA_FOO_DEPENDENCIES = бар
10:
11: LUA_FOO_BUILD_OPTS += BAR_INCDIR=$(STAGING_DIR)/usr/include
12: LUA_FOO_BUILD_OPTS += BAR_LIBDIR=$(STAGING_DIR)/usr/lib
13: LUA_FOO_LICENSE = лицензияluaFoo
14: LUA_FOO_LICENSE_FILES = $(LUA_FOO_SUBDIR)/КОПИРОВАНИЕ
15:
16: $(eval $(luarocks-package))
```

В строке 7 мы объявляем версию пакета (такую же, как в rockspec, которая представляет собой объединение версии upstream и ревизии rockspec, разделенных дефисом -).

В строке 8 мы объявляем, что пакет называется "foo" на LuaRocks. В Buildroot мы даем пакетам, связанным с Lua, имя, начинающееся с "Lua", поэтому имя Buildroot отличается от имени upstream. LUA\_FOO\_NAME\_UPSTREAM устанавливает связь между двумя именами.

В строке 9 мы объявляем наши зависимости от собственных библиотек, чтобы они были построены до начала процесса сборки нашего пакета.

В строках 11-12 мы говорим Buildroot передавать пользовательские параметры в LuaRocks при сборке пакета.

В строках 13-14 мы указываем установки для генерации пакета.

Наконец, в строке 16 мы вызываем макрос luarocks-package, который генерирует все правила Makefile, которые фактически позволяют собрать пакет.

Большинство этих данных можно получить из rock и rockspec. Таким образом, этот файл и файл Config.in можно сгенерировать, выполнив команду luarocks buildroot foo lua-foo в каталоге Buildroot. Эта команда запускает определенный аддон Buildroot для luarocks, который автоматически сгенерирует пакет Buildroot. Результат будет вручную проверить и, возможно, изменить.

- Файл package/Config.in необходимо обновить вручную, чтобы включить в него сгенерированные файлы Config.in.

### 18.10.2 luarocks- ссылка на пакет

LuaRocks — это система развертывания и управления для модулей Lua, которая поддерживает различные build.type: builtin, make и cmake. В контексте Buildroot инфраструктура luarocks-package поддерживает только построенный режим. Пакеты LuaRocks, использующие механизмы сборки make или cmake, должны быть упакованы с использованием инфраструктуры generic-package и cmake-package в Buildroot соответственно.

Основным макросом инфраструктуры пакета LuaRocks является luarocks-package: как и generic-package, он работает путем определения ряда переменных, предстающих метаданные о пакете, а затем вызова luarocks-package.

макрос .

Как и общая инфраструктура, инфраструктура LuaRocks работает путем определения ряда переменных перед вызовом макроса luarocks-package.

Все переменные метаданных пакета, которые используются в [общей инфраструктуре пакета](#), также используются в инфраструктуре LuaRocks.

Два из них заполняются инфраструктурой LuaRocks (для шага загрузки). Если ваш пакет не размещен на зеркале LuaRocks \$(BR2\_LUAROCKS\_MIRROR), вы можете переопределить их :

- LUA\_FOO\_SITE, по умолчанию \$(BR2\_LUAROCKS\_MIRROR)
- LUA\_FOO\_SOURCE, по умолчанию \$(LUA\_FOO\_NAME\_UPSTREAM в нижнем регистре)-\$(LUA\_FOO\_VERSION).src.rock

Также определены несколько дополнительных переменных, специфичных для инфраструктуры LuaRocks. В обычных случаях их можно переопределить.

- LUA\_FOO\_NAME\_UPSTREAM, по умолчанию lua-foo, т.е. имя пакета Buildroot
- LUA\_FOO\_ROCKSPEC, значение по умолчанию \$(LUA\_FOO\_NAME\_UPSTREAM в нижнем регистре)-\$(LUA\_FOO\_VERSION).rocksp
- LUA\_FOO\_SUBDIR, что по умолчанию равно \$(LUA\_FOO\_NAME\_UPSTREAM)-\$(LUA\_FOO\_VERSION\_WITHOUT\_ROCKSPEC)\_RE
- LUA\_FOO\_BUILD\_OPTS содержит дополнительные параметры сборки для вызова сборки luarocks.

### 18.11 Инфраструктура для пакетов Perl/CPAN

#### 18.11.1 Учебник по пакету perl

Сначала давайте посмотрим, как написать файл .mk для пакета Perl/CPAN, на примере:

```
01: ## ...
```

```
07: PERL_FOO_BAR_VERSION = 0.02 08: PERL_FOO_BAR_SOURCE
= Foo-Bar-${PERL_FOO_BAR_VERSION}.tar.gz 09: PERL_FOO_BAR_SITE = ${BR2_CPN_MIRROR}/authors/id/M/MO/MONGER 10:
PERL_FOO_BAR_DEPENDENCIES = perl-strictures 11: PERL_FOO_BAR_LICENSE = X удачес твенная или GPL-1.0+ 12:
PERL_FOO_BAR_LICENSE_FILES = Лиц евия 13: PERL_FOO_BAR_DISTNAME = Foo-Bar 14:
```

```
15: $(eval $(perl-package))
```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tarball и место расположение tarball на сервере CPAN. Buildroot автоматически загружает tarball из этого места расположения.

В строке 10 мы объявляем наши зависимости, чтобы они были построены до начала процесса сборки нашего пакета.

В строках 11 и 12 мы приводим лицензионные данные пакета (его лицензию в строке 11 и файл, содержащий текст лицензии, в строке 12).

В строке 13 указывается имя дистрибутива, необязательное с крипту utils/scanspan (для повторной генерации/обновления этих файлов пакета).

Наконец, в строке 15 мы вызываем макрос perl-package, который генерирует все правила Makefile, которые фактически позволяют обратиться к пакету.

Большую часть этих данных можно получить по адресу <https://metacpan.org/>. Итак, этот файл и Config.in можно сгенерировать, запустив с крипту utils/scanspan Foo-Bar в каталоге Buildroot (или в дереве br2-external). Этот скрипт создает файл Config.in и файл foo-bar.mk для запрошенного пакета, а также рекурсивно для всех зависимостей, указанных CPAN. Вам все равно следует вручную отредактировать результат. В частности, следует проверить следующее.

- Если модуль perl с введен с общей библиотекой, которая предстоит быть другим (не perl) пакетом, эта зависимость не добавляется автоматически. Ее необходимо добавить вручную в PERL\_FOO\_BAR\_DEPENDENCIES.
- Файл package/Config.in необходимо обновить вручную, чтобы включить сгенерированные файлы Config.in. В качестве подсказки с крипту scanspan выводит требуемые исходные "...утверждения отсортированные в алфавитном порядке.

### 18.11.2 Справочник по perl-пакету

В качестве политики все пакеты, предстоитющие модули Perl/CPAN, должны называться perl-<something> в Buildroot.

Эта инфраструктура обрабатывает различные системы сборки Perl: ExtUtils-MakeMaker (EUMM), Module-Build (MB) и Module-Build-Tiny. Build.PL предпочтительнее по умолчанию, когда пакет предстоит Makefile.PL и Build.PL.

Основной макрос инфраструктуры пакетов Perl/CPAN — perl-package. Он похож на макрос generic-package.

Возможность иметь целевые и составные пакеты также доступна с помощью макроса host-perl-package.

Как и общая инфраструктура, инфраструктура Perl/CPAN работает путем определения ряда переменных перед вызовом макроса perl-package.

Все переменные метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре Perl/CPAN.

Обратите внимание, что установка PERL\_FOO\_INSTALL\_STAGING в YES не имеет никакого эффекта, если не определена переменная PERL\_FOO\_INSTALL\_STAGING\_CMDS. Инфраструктура perl не определяет эти команды, поскольку модули Perl обычно не нужно устанавливать в промежуточный каталог.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры Perl/CPAN. Многие из них полезны только в очень специфических случаях, поэтому типичные пакеты будут использовать только некоторые из них.

- `PERL_FOO_PREFER_INSTALLER/HOST_PERL_FOO_PREFER_INSTALLER`, указывает предпочтительный метод установки.  
Возможные значения EUMM (для установки на основе Makefile.PL с использованием ExtUtils-MakeMaker) и MB (для установки на основе Build.PL с использованием Module-Build). Эта переменная используется только в том случае, если пакет предоставляет оба метода установки.
- `PERL_FOO_CONF_ENV/HOST_PERL_FOO_CONF_ENV`, чтобы указать дополнительные переменные среды для передачи в perl Makefile.PL или perl Build.PL. По умолчанию пусто.
- `PERL_FOO_CONF_OPTS/HOST_PERL_FOO_CONF_OPTS`, чтобы указать дополнительные параметры конфигурации для передачи в perl Makefile.PL или perl Build.PL. По умолчанию пусто.
- `PERL_FOO_BUILD_OPTS/HOST_PERL_FOO_BUILD_OPTS`, чтобы указать дополнительные параметры для передачи, чтобы сделать `pure_all` или perl Build с боркой на этапе сборки. По умолчанию пусто.
- `PERL_FOO_INSTALL_TARGET_OPTS`, чтобы указать дополнительные параметры для передачи при сборке `pure_install` или perl установить на этапе установки. По умолчанию пусто.
- `HOST_PERL_FOO_INSTALL_OPTS`, чтобы указать дополнительные параметры для передачи при сборке `pure_install` или perl установить на этапе установки. По умолчанию пусто.

## 18.12 Инфраструктура для виртуальных пакетов

В Buildroot виртуальный пакет — это пакет, функциональные возможности которого определяются одним или несколькими пакетами, называемыми поставщиками. Управление виртуальными пакетами — это расширяемый механизм, позволяющий пользователю выбирать поставщика, используемого в `rootfs`.

Например, OpenGL ES — это API для 2D- и 3D-графики на всех транзисторах с исключением. Реализация этого API отличается для платформ Allwinner Tech Sunxi и Texas Instruments OMAP3xx. Поэтому `libgles` будет виртуальным пакетом, а `sunxi-mali-utgard` и `ti-gfx` — поставщиками.

### 18.12.1 Учебник по виртуальным пакетам

В следующем примере мы объясняем, как добавить новый виртуальный пакет (`something-virtual`) и провайдера для него (`some-provider`).

Сначала давайте создадим виртуальный пакет.

### 18.12.2 Файл Config.in виртуального пакета

Файл `Config.in` виртуального пакета `something-virtual` должен содержать:

```
01: конфигурация BR2_PACKAGE_HAS_SOMETHING_VIRTUAL 02: логический
03:
04: конфигурация BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL 05:
      зависит от строки BR2_PACKAGE_HAS_SOMETHING_VIRTUAL
06:
```

В этом файле мы объявляем два параметра: `BR2_PACKAGE_HAS_SOMETHING_VIRTUAL` и `BR2_PACKAGE_PROVIDES_SOMETHING`, значения которых будут использоваться поставщиками.

### 18.12.3 Файл виртуального пакета .mk

Файл `.mk` для виртуального пакета должен просто определить макрос виртуального пакета:

```
01: ## ...
```

```
07: $(eval $(virtual-package))
```

Возможность иметь целиевые и хостовые пакеты также доступна с помощью макроса host-virtual-package.

#### 18.12.4 Файл Config.in поставщика

При добавлении пакета в качестве поставщика требуется явнонести некоторые изменения только в файл Config.in.

Файл Config.in пакета some-provider, который предоставляет функциональные возможности something-virtual, должен содержать:

```
01: config BR2_PACKAGE_SOME_PROVIDER 02: bool "some-provider"
select BR2_PACKAGE_HAS_SOMETHING_VIRTUAL help
03:     Это комментарий, который объясняет, что такое some-provider.
04:
05:
06:
07:     http://foosoftware.org/some-provider/
08:
09: если BR2_PACKAGE_SOME_PROVIDER 10: конфигурация
BR2_PACKAGE_PROVIDES_SOMETHING_VIRTUAL 11: по умолчанию "some-provider"
12: конец_конца
```

В строке 3 мы выбираем BR2\_PACKAGE\_HAS\_SOMETHING\_VIRTUAL, а в строке 11 мы устанавливаем значение BR2\_PACKAGE\_PROVIDES на имя провайдера, но только если он выбран.

#### 18.12.5 Файл .mk поставщика

Файл .mk также должен объявить дополнительную переменную SOME\_PROVIDER\_PROVIDES, содержащую имена всех виртуальных пакетов, реализацией которых он является

```
01: SOME_PROVIDER_PROVIDES = что-то виртуальное
```

Конечно, не забудьте добавить соответствующие зависимости с борки и времени выполнения для этого пакета!

#### 18.12.6 Примечания о зависимости от виртуального пакета

При добавлении пакета, которому требуется определенная функция, предоставляемая виртуальным пакетом, вам необходимо использовать зависимость от BR2\_PACKAGE\_HAS\_FEATURE, например так:

```
конфигурация BR2_PACKAGE_HAS_FEATURE логическое
значение
```

```
конфигурация BR2_PACKAGE_FOO
был "фу"
зависит от BR2_PACKAGE_HAS_FEATURE
```

### 18.12.7 Примечания о зависимости от конкретного поставщика

Если ваш пакет действительно требует определенного провайдера, то вам придется сделать свой пакет зависимым от этого провайдера; вы не можете выбрать провайдера.

Давайте рассмотрим пример с двумя поставщиками для FEATURE:

```
конфигурация BR2_PACKAGE_HAS_FEATURE логическое значение

конфигурация BR2_PACKAGE_FOO
    был "фу"
    выберите BR2_PACKAGE_HAS_FEATURE

конфигурация BR2_PACKAGE_BAR
    был "бар"
    выберите BR2_PACKAGE_HAS_FEATURE
```

И вы добавляете пакет, которому нужна FEATURE, предоставляемая foo, но не предоставляемая bar.

Если бы вы использовали select BR2\_PACKAGE\_FOO, то пользователь все равно мог бы выбрать BR2\_PACKAGE\_BAR в menuconfig. Это создало бы несогласование конфигурации, в результате чего оба поставщика одной и той же FEATURE были бы включены одновременно, один явно установлен пользователем, другой неявно вашим select.

Вместо этого вам придется использовать зависимость от BR2\_PACKAGE\_FOO, что позволит избежать неявной несогласованности конфигурации.

### 18.13 Инфраструктура для пакетов, использующих kconfig для файлов конфигурации

Популярным способом обработки пользовательской конфигурации программным пакетом является kconfig. Среди прочего, он используется ядром Linux, Busybox и самим Buildroot. Наличие файла .config и цели menuconfig — два известных признаков использования kconfig.

Buildroot предоставляет инфраструктуру для пакетов, которые используют kconfig для своей конфигурации. Эта инфраструктура обеспечивает необходящую логику для раскрытия цели menuconfig пакета как foo-menuconfig в Buildroot и для обработки и копирования файла конфигурации правильно образом.

Основной макрос инфраструктуры пакета kconfig — kconfig-package. Он похож на макрос generic-package.

Как и общая инфраструктура, инфраструктура kconfig работает путем определения ряда переменных перед вызовом kconfig-расмакроса.

Все переменные метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре kconfig.

Чтобы использовать инфраструктуру kconfig-package для пакета Buildroot, необходимо необходящие строки в файле .mk, в дополнение к переменным, требуемым инфраструктурой generic-package, следующие:

```
FOO_KCONFIG_FILE = ссылка на один файл конфигурации
$(eval $(kconfig-package))
```

Этот фрагмент создает следующие цели make:

- foo-menuconfig, который вызывает цель menuconfig пакета
- foo-update-config, который копирует конфигурацию обратно в исходный файл конфигурации. Это невозможно использовать цель при установке файлов фрагментов.
- foo-update-defconfig, который копирует конфигурацию обратно в исходный файл конфигурации. Файл конфигурации будет содержать только те параметры, которые отличаются от значений по умолчанию. Невозможно использовать эту цель, если установлены файлы фрагментов.

- foo-diff-config, который выводит различия между текущей конфигурацией и той, которая определена в конфигурации Buildroot для этого пакета kconfig. Вывод полезен для определения изменений конфигурации, которые, возможно, придется распространить на фрагменты конфигурации, например.

и обеспечивает копирование исходного файла конфигурации в каталог с борками в нужный момент.

Есть два варианта указания файла конфигурации для использования FOO\_KCONFIG\_FILE (как в примере выше) или FOO\_KCONFIG\_DEFCONFIG. Обязательно указать любой из них, но не оба:

- FOO\_KCONFIG\_FILE указывает путь к файлу defconfig или full-config, который будет использоваться для настройки пакета.
- FOO\_KCONFIG\_DEFCONFIG указывает правило defconfig make, которое следует вызывать для настройки пакета.

В дополнение к этим минимально необходимым строкам можно задать несколько дополнительных переменных в соответствии с потребностями распределения с матрицей пакета:

- FOO\_KCONFIG\_EDITORS: разделенный пробелами список поддерживаемых редакторов kconfig, например menuconfig xconfig. По умолчанию, menuconfig.
- FOO\_KCONFIG\_FRAGMENT\_FILES: разделенный пробелами список файлов фрагментов конфигурации, которые объединяются в основной файл конфигурации. Файлы фрагментов обычно используются когда есть желание оставаться с их интегрированными с высокоточным файлом конфигурации (def) с некоторыми незначительными изменениями.
- FOO\_KCONFIG\_OPTS: дополнительные параметры для передачи при вызове редакторов kconfig. Возможно, потребуется явно указать \$(FOO\_MAKE\_OPTS), например. По умолчанию пусто.
- FOO\_KCONFIG\_FIXUP\_CMDS: список команд оболочки, необходимых для исправления файла конфигурации после его копирования или запуска редактора kconfig. Такие команды могут быть необходимы для обеспечения соответствия конфигурации другим конфигурациям Buildroot, например. По умолчанию пусто.
- FOO\_KCONFIG\_DOTCONFIG: путь (с именем файла) к файлу .config относительно исходного дерева пакета. Значение по умолчанию .config должно хорошо подходить для всех пакетов, использующих стандартную инфраструктуру kconfig, установленную от ядра Linux; некоторые пакеты используют производную от kconfig, которая использует другое местоположение.
- FOO\_KCONFIG\_DEPENDENCIES: список пакетов (с корнем своего, пакетов хоста), которые необходимо обратить перед интерпретацией kconfig этого пакета. Используется редко. По умолчанию пусто.
- FOO\_KCONFIG\_SUPPORTS\_DEFCONFIG: поддерживает ли система kconfig пакета использование файлов defconfig; некоторые пакеты этого не делают. По умолчанию да

## 18.14 Инфраструктура для пакетов на основе арматуры

### 18.14.1 Учебное пособие по арматурному пакету

Сначала давайте посмотрим, как написать файл .mk для пакета на основе арматуры, на примере:

```
01: #####  
02: #  
03:#эрланг -фубар 04:#  
  
05: #####  
06:  
07: ERLANG_FOOBAR_VERSION = 1.0 08: ERLANG_FOOBAR_SOURCE  
= erlang-foobar-$(ERLANG_FOOBAR_VERSION).tar.xz 09: ERLANG_FOOBAR_SITE = http://www.foosoftware.org/download 10:  
ERLANG_FOOBAR_DEPENDENCIES = xocc t-libbbbb 11:  
  
12: $(eval $(rebar-package))
```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tar-архива (рекомендуется яхз-ед tar-архив) и местоположение tar-архива в Интернете.

Buildroot автоматически загрузит tarball из этого места.

В строке 10 мы объявляем наши зависимости, чтобы они были построены до начала процесса сборки нашего пакета.

Наконец, в строке 12 мы вызываем макрос `rebar-package`, который генерирует все правила `Makefile`, которые фактически позволяют собрать пакет.

## 18.14.2 Ссылка на пакет арматуры

Основной макрос инфраструктуры пакета арматуры — `rebar-package`. Он похож на макрос `generic-package`.

Возможно иметь хост-пакеты также доступна с помощью макроса `host-rebar-package`.

Как и общая инфраструктура, инфраструктура арматуры работает путем определения ряда переменных перед вызовом `rebar-package` макроса.

Все переменные метаданных пакета, которые определяются в [общей инфраструктуре пакета](#), также определяются в инфраструктуре арматуры.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры арматуры. Многие из них полезны только в очень специфических случаях, поэтому типичные пакеты будут использовать только некоторые из них.

- `ERLANG_FOOBAR_USE_AUTOCONF`, чтобы указать, что пакет использует `autoconf` на этапе конфигурации. Когда пакет устанавливается дляэтой переменной значение YES, он использует яинфраструктуру `autotools`.

**Примечание.** Вы также можете использовать некоторые переменные из инфраструктуры `autotools`: `ERLANG_FOOBAR_CONF_ENV`, `ERLANG_FOOBAR_ERLANG_FOOBAR_AUTORECONF`, `ERLANG_FOOBAR_AUTORECONF_ENV` и `ERLANG_FOOBAR_AUTORECONF_OPTS`.

- `ERLANG_FOOBAR_USE_BUNDLED_REBAR`, чтобы указать, что пакет имеет встроенную версию арматуры и что она должна быть использоваться. Допустимые значения ДА или НЕТ (по умолчанию).

**Примечание.** Если пакет включает утилиту для работы с арматурой, но может использовать универсальную, предоставляемую Buildroot, просто укажите НЕТ (т. е. не указывайте эту переменную). Установливайте только в том случае, если использование утилиты для работы с арматурой, в которой этот пакет, является обязательным.

- `ERLANG_FOOBAR_REBAR_ENV`, чтобы указать дополнительные переменные сюда для передачи в утилиту `rebar`.

- `ERLANG_FOOBAR_KEEP_DEPENDENCIES`, чтобы склонить зависимости, описанные в файле `rebar.config`. Допустимые значения YES или NO (по умолчанию). Если эта переменная установлена в YES, инфраструктура `rebar` удаляет такие зависимости в ходе после исправления, чтобы `rebar` не загружал и не компилировал их.

С инфраструктурой арматуры все шаги, необходимые для сборки и установки пакетов, уже определены, и они, как правило, хорошо работают для большинства пакетов на основе арматуры. Однако, при необходимости, все еще возможно настроить то, что делается я на каждом конкретном шаге:

- Добавляя постоперационный хук (после извлечения, исправления, установки, сборки или установки). Подробности см. в разделе [18.23](#).
- Переопределяя один из шагов. Например, даже если используется яинфраструктура арматуры, если файл `package.mk` определяется со своим собственным переменной `ERLANG_FOOBAR_BUILD_CMDS`, она будет использоваться вместе с переменной арматуры по умолчанию. Однако использование этого метода должно быть ограничено очень конкретными случаями. Не используйте его в общем случае.

## 18.15 Инфраструктура для пакетов на базе Waf

### 18.15.1 Учебник по waf-пакету

Сначала давайте посмотрим, как написать файл `.mk` для пакета на основе Waf, на примере:

```
01: ## ...
```

```
07: LIBFOO_VERSION = 1.0 08: LIBFOO_SOURCE  
= libfoo-${LIBFOO_VERSION}.tar.gz 09: LIBFOO_SITE = http://www.foosoftware.org/download 10:  
LIBFOO_CONF_OPTS = --enable-bar --disable-baz 11: LIBFOO_DEPENDENCIES = bar 12:
```

```
13: $(eval $(waf-пакет))
```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tar-архива (рекомендуется яxz-ed tar-архив) и место расположение tar-архива в Интернете. Buildroot автоматически загружает tarball из этого места.

В строке 10 мы сообщаем Buildroot, какие параметры следует включить для libfoo.

В строке 11 мы сообщаем Buildroot зависимости libfoo.

Наконец, в строке 13 мы вызываем макрос waf-package, который генерирует все правила Makefile, которые фактически позволяют собрать пакет.

## 18.15.2 Ссылка на waf-пакет

Основной макрос инфраструктуры пакета Waf — waf-package. Он похож на макрос generic-package.

Как и общая инфраструктура, инфраструктура Waf работает путем определения ряда переменных перед вызовом waf-package.

Все переменные метаданных пакета, которые упоминаются в [общей инфраструктуре пакета](#), также упоминаются в инфраструктуре Waf.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры Waf.

- LIBFOO\_SUBDIR может содержать имя подкаталога внутри пакета, содержащего основной файл wscript. Это полезно, если, например, основной файл wscript не находится в корне дерева, извлеченного из tarball. Если HOST\_LIBFOO\_SUBDIR не указан, по умолчанию используется LIBFOO\_SUBDIR.
- LIBFOO\_NEEDS\_EXTERNAL\_WAF может быть установлен на YES или NO, чтобы указать Buildroot использовать вложенный исполнимый файл waf. Если установлено значение NO (по умолчанию), то Buildroot будет использовать исполнимый файл waf, предоставленный в исходном дереве пакета; если установлено значение YES, то Buildroot загружает waf как хост-инструмент и будет использовать его для сборки пакета.
- LIBFOO\_WAF\_OPTS, чтобы указать дополнительные параметры для передачи в крипт waf на каждом этапе процесса сборки пакета: Настойка, сборка и установка. По умолчанию пусто.
- LIBFOO\_CONF\_OPTS, чтобы указать дополнительные параметры для передачи в крипт waf для шага конфигурации. По умолчанию пусто.
- LIBFOO\_BUILD\_OPTS, чтобы указать дополнительные параметры для передачи в крипт waf на этапе сборки. По умолчанию пусто.
- LIBFOO\_INSTALL\_STAGING\_OPTS, чтобы указать дополнительные параметры для передачи в крипт waf во время промежуточной установки шага. По умолчанию пусто.
- LIBFOO\_INSTALL\_TARGET\_OPTS, чтобы указать дополнительные параметры для передачи в крипт waf во время главной установки шага. По умолчанию пусто.

## 18.16 Инфраструктура для пакетов на основе Meson

### 18.16.1 Учебник по мезон-пакету

**Мезон** — это система сборки с открытым исходным кодом, призванная быть как чрезвычайно быстрой, так и, что еще важнее, максимально удобной для пользователя. Он использует [Ниндзя](#) как вспомогательный инструмент для выполнения реальных операций с сборкой.

Давайте рассмотрим, как написать файл .mk для пакета на основе Meson, на примере:

```

01: ##### #include <meson.h> #####
02: #
03: # фу
04: #
05: ##### #include <meson.h> #####
06: #
07: FOO_VERSION = 1.0 08: FOO_SOURCE
= foo-${FOO_VERSION}.tar.gz 09: FOO_SITE = http://www.foosoftware.org/
download 10: FOO_LICENSE = GPL-3.0+ 11: FOO_LICENSE_FILES = КОПИРОВАНИЕ

12: FOO_INSTALL_STAGING = ДА
13:
14: FOO_DEPENDENCIES = host-pkgconf bar 15:

16: ifeq ($(BR2_PACKAGE_BAZ),y)
17: FOO_CONF_OPTS += -Dbaz=истина
18: FOO_DEPENDENCIES += баз 19: иначе

20: FOO_CONF_OPTS += -Dbaz=false 21: конец_файла

22:
23: $(eval $(meson-package))

```

Makefile начинается с определения стандартных переменных для объявления пакета (строки 7–11).

В строке 23 мы вызываем макрос `meson-package`, который генерирует все правила Makefile, которые фактически позволяют собрать пакет.

В этом примере `host-pkgconf` и `bar` объявлены как зависимости в `FOO_DEPENDENCIES` в строке 14, поскольку файл с сборкой Meson `foo` использует `pkg-config` для определения флагов компиляции и библиотек пакета `bar`.

Обратите внимание, что нет необходимости добавлять `host-meson` в переменную `FOO_DEPENDENCIES` пакета, поскольку эта базовая зависимость автоматически добавляется в строке 17, как указано в файле `meson_options.txt` в исходном дереве "foo". Пакет "baz" также добавляется явно в `FOO_DEPENDENCIES`. Обратите внимание, что поддержка `baz` явно отключена в строке 20, если пакет не выбран.

Подводя итог, можно сказать, что для добавления нового пакета на основе Meson пример Makefile можно скопировать дословно, а затем отредактировать, заменив все вхождения `FOO` на имя нового пакета в верхнем регистре и обновив значения стандартных переменных.

### 18.16.2 Ссылка на мезонный пакет

Основной макрос инфраструктуры пакета Meson — `meson-package`. Он похож на макрос `generic-package`.

Можно иметь целевые и составные пакеты также доступны с помощью макроса `host-meson-package`.

Как и общая инфраструктура, инфраструктура Meson работает путем определения ряда переменных перед вызовом `meson-package` макроса.

Все переменные информации метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре Meson.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры Meson. Многие из них полезны только с специфическими случаями, поэтому типичные пакеты будут использовать только некоторые из них.

- FOO\_SUBDIR может содержать имя подкаталога внутри пакета, содержащего основной файл meson.build. Это полезно, если, например, основной файл meson.build не находится в корне дерева, извлеченного из tarball. Если HOST\_FOO\_SUBDIR не указан, по умолчанию используется FOO\_SUBDIR.
- FOO\_CONF\_ENV, чтобы указать дополнительные переменные с ready для передачи в meson для шага конфигурации. По умолчанию пусто.
- FOO\_CONF\_OPTS, чтобы указать дополнительные параметры для передачи meson на этапе конфигурации. По умолчанию пусто.
- FOO\_CFLAGS, для указания аргументов компилятора, добавленных в список c\_args файла cross-compile.conf, специфичное для пакета. По умолчанию значение TARGET\_CFLAGS.
- FOO\_CXXFLAGS, для указания аргументов компилятора, добавленных в список cpp\_args файла cross-compile.conf, специфичное для пакета. По умолчанию значение TARGET\_CXXFLAGS.
- FOO\_LDFLAGS, для указания аргументов компилятора, добавленных в списки c\_link\_args и cpp\_link\_args файла cross-compile.conf, специфичные для пакета. По умолчанию значение TARGET\_LDFLAGS.
- FOO\_MESON\_EXTRA\_BINARIES, чтобы указать разделенный пробелами список программ для добавления в раздел [binaries] файла конфигурации meson cross-compilation.conf. Формат — program-name='path/to/program', без пробела вокруг знака = и с одинарными кавычками вокруг строковых значений. По умолчанию пусто. Обратите внимание, что Buildroot уже устанавливает правильные значения для needs\_exe\_wrapper, c\_args, c\_link\_args, cpp\_args, cpp\_link\_args, sys\_root и pkg\_config libdir.
- FOO\_MESON\_EXTRA\_PROPERTIES, чтобы указать разделенный пробелами список свойств для добавления в раздел [properties] файла конфигурации meson cross-compilation.conf. Формат — property-name=<value> без пробела вокруг знака = и с одинарными кавычками вокруг строковых значений. По умолчанию пусто. Обратите внимание, что Buildroot уже задает значения для needs\_exe\_wrapper, c\_args, c\_link\_args, cpp\_args, cpp\_link\_args, sys\_root и pkg\_config libdir.
- FOO\_NINJA\_ENV, чтобы указать дополнительные переменные с ready для передачи в ninja, с отсутствующим инструментом meson, отвечающим за сборки. По умолчанию пусто.
- FOO\_NINJA\_OPTS, чтобы указать разделенный пробелами список целей для построения. По умолчанию пусто, чтобы построить цель(и) по умолчанию.

## 18.17 Инфраструктура для грузовых пакетов

Cargo — это менеджер пакетов для языка программирования Rust. Он позволяет пользователю сортировать программы или библиотеки, написанные на Rust, но также управляет их зависимостями, чтобы обеспечить повторяемость сборки. Пакеты Cargo называются «ящиками».

### 18.17.1 Учебник по грузовым пакетам

Файл Config.in пакета foo на базе Cargo должен содержать:

```

01: конфигурация BR2_PACKAGE_FOO 02:
    был "ф"
03: зависит от BR2_PACKAGE_HOST_RUSTC_TARGET_ARCH_SUPPORTS выберите
04: BR2_PACKAGE_HOST_RUSTC help Это комментарий,
05:
06: который объясняет, что такое foo.
07:
08: http://foosoftware.org/foo/

```

А файл .mk для этого пакета должен содержать:

```

01: ##### #####
02: #
03: # ф
04: #
05: ##### #####
06:
07: FOO_VERSION = 1.0

```

```
08: FOO_SOURCE = foo-${FOO_VERSION}.tar.gz 09: FOO_SITE = http://
www foosoftware.org/download 10: FOO_LICENSE = GPL-3.0+ 11: FOO_LICENSE_FILES = КОПИРОВАНИЕ
12: 13: $(eval $(cargo-package))
```

Makefile начинается с определения стандартных переменных для объявления пакета (с троек 7-11).

Как видно из троек 13, он основан на инфраструктуре грузовогопакета. Груз будет автоматически вызван этой инфраструктурой для сборки и установки пакета.

По-прежнему можно определять пользовательские команды сборки или установки (например, с помощью FOO\_BUILD\_CMDS и FOO\_INSTALL\_TARGET). Они затем заменят команды из инфраструктуры Cargo.

## 18.17.2 Ссылка на грузовое место

Основными макросами для инфраструктуры грузовых пакетов являются cargo-package для целевых пакетов и host-cargo-package для основных пакетов.

Как и общая инфраструктура, инфраструктура Cargo работает путем определения ряда переменных перед вызовом макросов cargo-package или host-cargo-package.

Все переменные метаданных пакета, которые существоуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре Cargo. закон.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры Cargo. Многие из них полезны только в очень специфических случаях, поэтому типичные пакеты будут использовать только некоторые из них.

- FOO\_SUBDIR может содержать имя подкаталога внутри пакета, содержащего файл Cargo.toml. Это полезно, если, например, он не находится в корне дерева, извлеченного из tarball. Если HOST\_FOO\_SUBDIR не указан, по умолчанию используется FOO\_SUBDIR.
- FOO\_CARGO\_ENV может использоваться для передачи дополнительных переменных в среде вызовов грузов. Он используется как при сборке и временная установка.
- FOO\_CARGO\_BUILD\_OPTS можно использовать для передачи дополнительных параметров грузу во время сборки.
- FOO\_CARGO\_INSTALL\_OPTS можно использовать для передачи дополнительных параметров грузу во время установки.

Язык может зависеть от других библиотек из репозиториев crates.io или git, перечисленных в его файле Cargo.toml. Buildroot автоматически берет на себя загрузку таких зависимостей как часть этапа загрузки пакетов, используя инфраструктуру cargo-package. Затем такие зависимости вместе с исходным кодом пакета в tarball, скомпилированном в DL\_DIR Buildroot, и, следовательно, эштарбэлл пакета охватывает не только источник пакета, но и источники зависимостей. Таким образом, изменение, введенное в одну из зависимостей, также будет обнаружено проверкой эша. Кроме того, этот механизм позволяет выполнять сборку полностью в автономном режиме, поскольку cargo не будет выполнять никаких загрузок во время сборки. Этот механизм называется явендорингом зависимостей.

## 18.18 Инфраструктура для пакетов Go

Эта инфраструктура применяется к пакетам Go, которые используют стандартную систему сборки и используют вложенные зависимости.

### 18.18.1 golang-package учебник

Сначала давайте посмотрим, как написать файл .mk для пакета go, на примере:

```

01: ## ...

07: FOO_VERSION = 1.0 08: FOO_SITE
= ${вызов github,bar,foo,$(FOO_VERSION)}
09: FOO_LICENSE = BSD-3-Лицензия 10: FOO_LICENSE_FILES =
ЛИЦЕНЗИЯ 11:

12: $(eval $(golang-пакет))

```

В строке 7 мы объявляем версию пакета.

В строке 8 мы объявляем исходное расположение пакета, в данном случае взятое из Github, поскольку большое количество пакетов Go размещено на Github.

В строках 9 и 10 мы приводим лицензионные данные о пакете.

Наконец, в строке 12 мы вызываем макрос golang-package, который генерирует все правила Makefile, которые фактически позволяют собрать пакет.

## 18.18.2 Справочник по golang-пакету

В файле Config.in пакеты, использующие инфраструктуру golang-package, должны зависеть от BR2\_PACKAGE\_HOST\_GO\_TARG, поскольку Buildroot автоматически добавит зависимость от host-go к таким пакетам. Если вам нужна поддержка CGO в вашем package-age, вы должны добавить зависимость от BR2\_PACKAGE\_HOST\_GO\_TARGET\_CGO\_LINKING\_SUPPORTS; для пакетов host-golang-package, должны зависеть от BR2\_PACKAGE\_HOST\_GO\_HOST\_CGO\_LINKING\_SUPPORTS.

Основной макрос инфраструктуры пакетов Go — golang-package. Он похож на макрос generic-package.

Также доступна возможность сборки пакетов host-golang-package с помощью макроса host-golang-package. Пакеты host-golang-package, должны зависеть от BR2\_PACKAGE\_HOST\_GO\_HOST\_ARCH\_SUPPORTS.

Как и общая инфраструктура, инфраструктура Go работает путем определения ряда переменных перед вызовом golang-package макроса.

Все переменные метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре Go.

Обратите внимание, что нет необходимости добавлять host-go в переменную FOO\_DEPENDENCIES пакета, поскольку эта базовая зависимость автоматически добавляется по мере необходимости инфраструктурой пакета Go.

Несколько дополнительных переменных, специфичных для инфраструктуры Go, могут быть определены дополнительно в зависимости от потребностей пакета. Многие из них полезны только очень специфических случаях, поэтому типичные пакеты будут использовать только несколько из них или не будут использовать ни одной.

- Пакет должен указывать имя модуля Go в переменной FOO\_GOMOD. Если не указано, по умолчанию используется URL-domain/1st-part, например, FOO\_GOMOD примет значение github.com/bar/foo для пакета, который указывает FOO\_SITE = \$(call github,bar). Инфраструктура пакета Go автоматически генерирует минимальный файл go.mod в исходном дереве пакета, если он не существует.
- FOO\_LDFLAGS и FOO\_TAGS можно использовать для передачи LDFLAGS или TAGS соответственно в команду go build.
- FOO\_BUILD\_TARGETS может использоваться для передачи списков целей, которые должны быть построены. Если FOO\_BUILD\_TARGETS не указан, по умолчанию это ... Тогда есть два случая
  - FOO\_BUILD\_TARGETS – это .. В этом случае мы предполагаем, что будет создан только один двоичный файл, и что по умолчанию мы назовем его именем пакета. Если это не подходит, имя полученного двоичного файла можно переопределить с помощью FOO\_BIN\_NAME.
  - FOO\_BUILD\_TARGETS не является .. В этом случае мы выбираем значения для борьбы с каждой целью и для каждой сгенерированного двоичного файла, который является компонентом цели, не являющимся каталогом. Например, если FOO\_BUILD\_TARGETS = cmd/docker cmd/dockerd, то создаваемые двоичные файлы — это docker и dockerd.
- FOO\_INSTALL\_BINS можно использовать для передачи списков двоичных файлов, которые должны быть установлены в /usr/bin на цели. Если FOO\_INSTALL\_BINS не указан, по умолчанию используется имя пакета в нижнем регистре.

С инфраструктурой Go все шаги, необязательные для сборки и установки пакетов, уже определены, и они, как правило, хорошо работают для большинства пакетов на основе Go. Однако, при необходимости, все еще возможно настроить то, что делается на каждом конкретном шаге:

- Добавляя пост-операционный хук (после извлечения, исправления, настройки, сборки или установки). Подробности см. в разделе [18.23](#).
- Переопределяя один из шагов. Например, даже если используется ядро инфраструктуры Go, если файл пакета.mk определяет собственную переменную FOO\_BUILD\_CMDS, она будет использоваться вместо переменной Go по умолчанию. Однако использование этого метода должно быть ограничено очень конкретными случаями. Не используйте его в общем случае.

Пакет Go может зависеть от других модулей Go, перечисленных в его файле go.mod. Buildroot автоматически берет на себя загрузку таких зависимостей как часть этапа загрузки пакетов, используя инфраструктуру golang-package. Такие зависимости затем соединяются вместе с их одним кодом пакета в tarball, кэшированном в DL\_DIR Buildroot, и, следовательно, этот tarball пакет включает такие зависимости.

Этот механизм гарантирует обнаружение любых изменений в зависимостях и позволяет выполнять сборку полностью в автономном режиме.

## 18.19 Инфраструктура для пакетов на основе QMake

### 18.19.1 Учебник по qmake-package

Сначала давайте посмотрим, как написать файл .mk для пакета на основе QMake, например:

```
01: #####  
02: #  
03: # libfoo  
04: #  
05: #####  
06:  
07: LIBFOO_VERSION = 1.0.0  
LIBFOO_SOURCE = libfoo-$LIBFOO_VERSION.tar.gz  
09: LIBFOO_SITE = http://  
www foosoft.org/download  
10: LIBFOO_CONF_OPTS = QT_CONFIG+=bar QT_CONFIG-  
=baz  
11: LIBFOO_DEPENDENCIES = bar  
12:  
13: $(eval $(qmake-package))
```

В строке 7 мы объявляем версию пакета.

В строках 8 и 9 мы объявляем имя tar-архива (рекомендуется xz-ed tar-архив) и местоположение tar-архива в Интернете.

Buildroot автоматически загрузит tarball из этого места.

В строке 10 мы сообщаем Buildroot, какие параметры следует включить для libfoo.

В строке 11 мы сообщаем Buildroot зависимости libfoo.

Наконец, в строке 13 мы вызываем макрос qmake-package, который генерирует все правила Makefile, которые фактически позволяют собрать пакет.

### 18.19.2 Ссылка на qmake-пакет

Основной макрос инфраструктуры пакета QMake — qmake-package. Он полагается на макрос generic-package.

Как и общая инфраструктура, инфраструктура QMake работает путем определения ряда переменных перед вызовом qmake-package.

Все переменные метаданных пакета, которые существуют в [общей инфраструктуре пакета](#), также существуют в инфраструктуре QMake.

Также можно определить несколько дополнительных переменных, специфичных для инфраструктуры QMake.

- LIBFOO\_CONF\_ENV, чтобы указать дополнительные переменные с реды для передачи в скрипту qmake для этапа конфигурации. По умолчанию, пусто.
- LIBFOO\_CONF\_OPTS, чтобы указать дополнительные параметры для передачи скрипту qmake для шага конфигурации. По умолчанию пусто.
- LIBFOO\_MAKE\_ENV, чтобы указать дополнительные переменные с реды для команды make во время этапов сборки и установки. По умолчанию пусто.
- LIBFOO\_MAKE\_OPTS, чтобы указать дополнительные цели для передачи команде make на этапе сборки. По умолчанию пусто.
- LIBFOO\_INSTALL\_STAGING\_OPTS, чтобы указать дополнительные цели для передачи команде make во время промежуточной установки шага installation. По умолчанию установить.
- LIBFOO\_INSTALL\_TARGET\_OPTS, чтобы указать дополнительные цели для передачи команде make во время установки цели шага. По умолчанию установить.
- LIBFOO\_SYNC\_QT\_HEADERS, для запуска syncqt.pl перед qmake. Некоторым пакетам это нужно для правильно заполненного include каталога перед загрузкой сборки.

## 18.20 Инфраструктура для пакетов, создавающих модули ядра

Buildroot предлагает вспомогательную инфраструктуру, которая упрощает написание пакетов, которые собирают и устанавливают модули ядра Linux. Некоторые пакеты содержат только модуль ядра, другие пакеты содержат программы и библиотеки в дополнение к модулям ядра. Вспомогательная инфраструктура Buildroot поддерживает оба случая.

### 18.20.1 Учебник по модулю ядра

Начнем с примера подготовки простого пакета, который собирает только модуль ядра и никаких других компонентов:

```
01: #####  
02: #  
03: # фу  
04: #  
05: #####  
06:  
07: FOO_VERSION = 1.2.3 08: FOO_SOURCE  
= foo-${FOO_VERSION}.tar.xz 09: FOO_SITE = http://www.foosoftware.org/  
download 10: FOO_LICENSE = GPL-2.0  
  
11: FOO_LICENSE_FILES = КОПИРОВАНИЕ  
12:  
13: $(eval $(modуль-ядра)) 14: $(eval $  
(универсальный-пакет))
```

Строки 7-11 определяют обычные метаданные для указания версии, имени архива, удаленного URI, по которому можно найти источник пакета, и информации о лицензировании.

В строке 13 мы вызываем вспомогательную инфраструктуру модуля ядра, которая генерирует все соответствующие правила и переменные Makefile для сборки этого модуля ядра.

Наконец, в строке 14 мы вызываем [инфраструктуру универсального пакета](#).

Зависимость от Linux добавляется автоматически, поэтому указывать ее в FOO\_DEPENDENCIES не нужно.

Вы могли заметить, что в отличие от других инфраструктур пакетов мы явно вызываем вторую инфраструктуру. Это позволяет пакету построить модуль ядра, но также, при необходимости, использовать любую из других инфраструктур пакетов для сборки обычных компонентов пользовательского просмотра (библиотек, исключительных файлов...). Использование инфраструктур модуля ядра само по себе недостаточно; необходимо использовать другую инфраструктуру пакета.

Давайте рассмотрим более ложный пример:

```

01: ## ...

07: FOO_VERSION = 1.2.3 08: FOO_SOURCE =
foo-$(FOO_VERSION).tar.xz 09: FOO_SITE = http://www.foosoftware.org/download 10:
FOO_LICENSE = GPL-2.0 11: FOO_LICENSE_FILES = КОПИРОВАНИЕ 12:

13: FOO_MODULE_SUBDIRS = драйвер/база
14: FOO_MODULE_MAKE_OPTS = KVERSION=$(LINUX_VERSION_PROBED) 15:

16: ifeq ($(BR2_PACKAGE_LIBBAR),y)
17: FOO_DEPENDENCIES += libbar
18: FOO_CONF_OPTS += --enable-bar
19: FOO_MODULE_SUBDIRS += драйвер/бар 20: иначе

21: FOO_CONF_OPTS += --disable-bar 22: endif

23:
24: $(eval $(модуль-ядра)) 26: $(eval $(пакет-
autotools))

```

Здесь мы видим, что у нас есть пакет на основе autotools, который также собирает модуль ядра, расположенный в подкаталоге driver/base, и если включена библиотека libbar, модуль ядра, расположенный в подкаталоге driver/bar, и определяет переменную KVERSION, которая будет передана в систему с борками Linux при сборке модуля(ей).

## 18.20.2 Ссылка на модуль ядра

Основной макрос для инфраструктуры модуля ядра — kernel-module. В отличие от других инфраструктур пакетов, он не является автономным и требует вызова вог о другого макроса \*-package после него.

Макрос kernel-module определяет шаги post-build и post-target-install для сборки модулей ядра. Если .mk пакета нужен дистрибутик с обработанным модулем ядра, он должен с делать это в ходе post-build, зарегистрированном после вызова kernel-module.

Аналогично, если .mk пакета нуждается в дистрибутике к модулю ядра после его установки, он должен с делать это в ходе post-install, зарегистрированном после вызова kernel-module. Вот пример:

```

$(eval $(модуль-ядра))

define FOO_DO_STUFF_WITH_KERNEL_MODULE # Сделать с
    этим что-нибудь...
    опуск тильда
FOO_POST_BUILD_HOOKS += FOO_DO_STUFF_WITH_KERNEL_MODULE

$(eval $(generic-package))

```

Наконец, в отличие от других инфраструктур пакетов, не существует варианта с тядро-модуль для построения модулях ос тядра.

Для дальнейшей настройки с борками модуля ядра можно дополнительного определить следующие переменные:

- FOO\_MODULE\_SUBDIRS может быть установлен в один или несколько подкаталогов (относительно верхнего каталога исходного кода пакета), где находятся ядры модуля ядра. Если он пуст или не установлен, исходные коды для модуля(ей) ядра считаются расположеными в верхней части дерева исходного кода пакета.
- FOO\_MODULE\_MAKE\_OPTS может быть настроен на включение дополнительных определений переменных для передачи в систему с борками Linux.

Вы также можете ссылаться (но не устанавливать!) на эти переменные:

- LINUX\_DIR содержит путь к месту, где было извлечено и собрано ядро Linux.
- LINUX\_VERSION содержит строку версии, настроенную пользователем.
- LINUX\_VERSION\_PROBED содержит реальную строку версии ядра, полученную с помощью команды make -C \$(LINUX\_DIR) rel diz ядра
- KERNEL\_ARCH содержит название текущей архитектуры, например arm, mips...

## 18.21 Инфраструктура для документов asciidoc

Руководство по Buildroot, которое вы сейчас читаете, полностью написано с использованием [AsciiDoc](#) с интаксис разметки. Затем руководство преобразуется в множество форматов:

- HTML-код
- с плит-html
- pdf
- epub
- текст

Хотя Buildroot содержит только один документ, написанный на языке AsciiDoc, для пакетов существует инфраструктура для рендеринга документов с использованием интаксиса AsciiDoc.

Также, как и для пакетов, инфраструктура AsciiDoc доступна из дерева br2-external. Это позволяет документации для дерева br2-external соответствовать документации Buildroot, поскольку она будет отображаться в тех же форматах и с использованием той же компоновки и темы.

### 18.21.1 asciidoc-документ учебник

В то время как инфраструктуры пакетов имеют суффикс -package, инфраструктуры документов имеют суффикс -document. Таким образом, инфраструктура AsciiDoc называется яasciidoc-document.

Вот пример визуализации просмотра документа AsciiDoc.

```
01: ## ...
```

```
07: FOO_SOURCES = $(sort $(wildcard $(FOO_DOCDIR)/*)) 08: $(eval $(call asciidoc-document))
```

В строке 7 Makefile объявляет, какие источники документа. В это время ожидается, что источники документа будут только локальными; Buildroot не будет пытаться яничего загружать для рендеринга документа. Таким образом, вы должны указать, где находятся источники. Обычно строка выше достаточно для документа без структуры подкаталогов.

В строке 8 мы вызываем функцию asciidoc-document, которая генерирует весь код Makefile, необходимый для визуализации документа.

### 18.21.2 asciidoc- ссылка на документ

Список переменных, которые можно задать в файле .mk для представления метаданных (предполагается что имя документа — foo):

- FOO\_SOURCES, обязательный, определяет исходные файлы для документа.
- FOO\_RESOURCES, не обязательно, может содержать разделенный пробелами список путей к одному или нескольким каталогам, содержащим так называемые Ресурсы (например, CSS или изображения). По умолчанию пустые.
- FOO\_DEPENDENCIES, не обязательно, с список пакетов (с корнем всего, хост-пакетов), которые должны быть собраны перед сборкой этого документа.
- FOO\_TOC\_DEPTH, FOO\_TOC\_DEPTH\_<FMT>, необязательные параметры, глубина таблицы с содержания для этого документа, которая может быть определена для указанного формата <FMT> (см. список отображаемых форматов выше, но вверх не регистре и с заменой тире на подчеркивание; см. пример ниже). По умолчанию: 1.

Существуют также дополнительные hooks (общую информацию о них см. в разделе 18.23), которые документ может установить для определения дополнительных действий, которые необходимо выполнить на различных этапах:

- FOO\_POST\_RSYNC\_HOOKS для запуска дополнительных команд после копирования исходников Buildroot. Это может быть использовано, например, для генерации части руководства с информацией, извлеченной из дерева. Например, Buildroot использует этот hook для генерации таблиц в приложениях.
- FOO\_CHECK\_DEPENDENCIES\_HOOKS для запуска дополнительных тестов на требуемых компонентах для генерации документа. В AsciiDoc можно вызывать фильтры, то есть программы, которые будут анализировать блок AsciiDoc и отображать его соответствующим образом (например, ditaa или аффигура).
- FOO\_CHECK\_DEPENDENCIES\_<FMT>\_HOOKS, для запуска дополнительных тестов для указанного формата <FMT> (см. список обновлений) (см. выше) (форматы, указанные выше).

Buildroot устанавливает следующую переменную, которую можно использовать в определениях выше:

- \${FOO\_DOCDIR}, аналогично \${FOO\_PKGDIR}, содержит путь к каталогу, с содержащему foo.mk. Его можно использовать для ссылки на источники документа, а также в уках, основанных на post-rsync, если необходимо сгенерировать части документации.
- \${@D}, как и для традиционных пакетов, содержит путь к каталогу, в который будет скопирован и собран документ.

Вот полный пример, в котором используется множество переменных и их уки:

```

01: ##### 02: #
03: # foo-документ
04: #
05: ##### 06:
07: FOO_SOURCES = $(определка $(подстановочный знак ${FOO_DOCDIR}/*))
08: FOO_RESOURCES = $(определка $(подстановочный знак ${FOO_DOCDIR}/ресурсы)) 09:
09:
10: FOO_TOC_DEPTH = 2
11: FOO_TOC_DEPTH_HTML = 1
12: FOO_TOC_DEPTH_SPLIT_HTML = 3 13:
13:
14: определить FOO_GEN_EXTRA_DOC /путь/
    сгенерированному -с крипту --outdir=${@D} 15:
15: уйти
16:
17: FOO_POST_RSYNC_HOOKS += FOO_GEN_EXTRA_DOC
18:
19: определение FOO_CHECK_MY_PROG
20:     если ! which my-prog >/dev/null 2>&1; тогда \

```

```

21:         echo "Вам понадобится яму-prog для генерации документа foo"; \
22:         вых од; \
23:         быть
24: уйти
25: FOO_CHECK_DEPENDENCIES_HOOKS += FOO_CHECK_MY_PROG
26:
27: определить FOO_CHECK_MY_OTHER_PROG
28:     если ! which my-other-prog >/dev/null 2>&1; тогда \
29:         echo "Вам понадобится яму-other-prog для генерации документа foo в формате PDF"; \
30:         вых од; \
31:         быть
32: уйти
33: FOO_CHECK_DEPENDENCIES_PDF_HOOKS += FOO_CHECK_MY_OTHER_PROG
34:
35: $(eval $(call asciidoc-document))

```

## 18.22 Инфраструктура, с помощью которой для пакета ядра Linux

Пакет ядра Linux может использовать некоторые специальные инфраструктуры, основанные на пакетных хуках, для создания инструментов ядра Linux и/или создание расширений ядра Linux.

### 18.22.1 linux-kernel-инструменты

Buildroot предлагает вам использовать инфраструктуру для создания некоторых инструментов пользователей для распространения для целевой платформы, состоящих из одних кодах ядра Linux. Поскольку их исходный код является частью исходного кода ядра, существует специальный пакет `linux-tools`, который повторно использует исходные коды ядра Linux, работающие на целевой машине.

Давайте рассмотрим пример инструмента Linux. Для нового инструмента Linux с именем `foo` создайте новый пункт меню в существующем `package/linux-tools`. Этот файл будет содержать описание параметров, связанных с каждым инструментом ядра, которые будут использоваться и отображаться в инструменте конфигурации. В принципе это будет выглядеть так:

```

01: конфигурация BR2_PACKAGE_LINUX_TOOLS_FOO
02:     был "фу"
03:     выберите BR2_PACKAGE_LINUX_TOOLS
04:     помощь
05:         Это комментарий, который объясняет, что такое инструмент ядра foo.
06:
07:         http://foosoftware.org/foo/

```

Имя опции начинается с префикса `BR2_PACKAGE_LINUX_TOOLS_`, за которым следуют заглавное имя инструмента.  
(подобному, как это делается для пакетов).

Примечание. В отличие от других пакетов, параметры пакета `linux-tools` отображаются в меню ядра Linux, в разделе «Ядро Linux». Подменю «Инструменты», а не главное меню «Целевые пакеты».

Затем для каждого инструмента Linux добавьте новый файл `.mk.in` с именем `package/linux-tools/linux-tool-foo.mk.in`. Это будет в новом выглядеть так:

```

01: ##### #####
02: #
03: # фу
04: #
05: ##### #####
06:
07: LINUX_TOOLS += foo
08:
09: FOO_DEPENDENCIES = libbbb
10:

```

```

11: определение FOO_BUILD_CMDS 12:
    $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools foo
13: ути
14:
15: определение FOO_INSTALL_STAGING_CMDS
    $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \ 16:
17:         DESTDIR=$(STAGING_DIR) \
18:             foo_install
19: ути
20:
21: определение FOO_INSTALL_TARGET_CMDS
22:     $(TARGET_MAKE_ENV) $(MAKE) -C $(LINUX_DIR)/tools \
23:         DISTDIR=$(TARGET_DIR) \ foo_install
24:
25: ути

```

В строке 7 мы регистрируем инструмент Linux foo в список доступных инструментов Linux.

В строке 9 мы указываем список зависимостей, на которые опирается этот инструмент. Эти зависимости добавляются в список зависимостей пакета Linux только при выборе инструмента foo.

Остальная часть Makefile, с строками 11-25, определяет, что должно быть сделано на разных этапах процесса сборки инструмента Linux, как для универсального пакета. Фактически они будут использоваться только при выборе инструмента foo. Поддерживаются только команды \_BUILD\_CMDS, \_INSTALL\_STAGING\_CMDS и \_INSTALL\_TARGET\_CMDS.

**Примечание.** Нельзя вызывать \${eval \$(generic-package)} или любую другую инфраструктуру пакетов! Инструменты Linux не являются пакетами сами по себе, они являются частью пакета linux-tools.

## 18.22.2 Linux-ядро-расширения

Некоторые пакеты предоставляют новые функции, требующие изменения дерева ядра Linux. Это может быть в форме исключений, которые должны быть применены к дереву ядра, или в форме новых файлов, которые должны быть добавлены в дерево. Инфраструктура расширений ядра Linux Buildroot предоставляет простое решение для автоматического выполнения этого сразу после извлечения исходных кодов ядра и до применения исключений ядра. Примерами расширений, упакованных с использованием этого механизма, являются ядра расширения реального времени Xenomai и RTAI, а также набор драйверов ЖК-экранов из дерева fbfft.

Давайте рассмотрим пример добавления нового расширения Linux foo.

Сначала создайте пакет foo, который предоставляет расширение: этот пакет является стандартным пакетом; см. предыдущие главы о том, как создать такой пакет. Этот пакет отвечает за загрузку архива исходников, проверку его, определение информации олицензии и создание инструментов пользователя для просмотра трансфера, если таковые имеются.

Затем создайте расширение Linux как такое: создайте новую запись меню в существующем linux/Config.ext.in. Этот файл содержит описание опций, связанных с каждым расширением ядра, которые будут использоваться и отображаться в инструменте конфигурации. В основном это будет выглядеть так:

```

01: конфигурация BR2_LINUX_KERNEL_EXT_FOO 02:
    был "foo"
03: help
04:     Это комментарий, который объясняет, что такое расширение ядра foo.
05:
06:     http://foosoftware.org/foo/

```

Затем для каждого расширения linux добавьте новый файл .mk с именем linux/linux-ext-foo.mk. Он должен содержать:

```
01: ## ...
```

```
07: LINUX_EXTENSIONS += foo
```

```
08:  
09: определение FOO_PREPARE_KERNEL 10:  
    ${FOO_DIR}/prepare-kernel-tree.sh --linux-dir=$(@D)  
11: уйти
```

В строке 7 мы добавляем расширение Linux foo в список доступных расширений Linux.

В строках 9–11 мы определяем, что должно быть сделано расширением для изменения дерева ядра Linux; это относится только к расширению Linux и может использовать переменные, определенные пакетом foo, например: \${FOO\_DIR} или \${FOO\_VERSION}... а также все переменные Linux, например: \${LINUX\_VERSION} или \${LINUX\_VERSION\_PROBED}, \${KERNEL\_ARCH}... Смотрите [определение этих переменных ядра](#).

## 18.23 Крючки, доступные на разных этапах сборки

Общая инфраструктура (и, как следствие, производные инфраструктуры autotools и cmake) позволяет пакетам указывать hooks. Они определяют дальнейшие действия, которые необходимо выполнить после успешного шага. Большинство твоих уков не очень полезны для универсальных пакетов, поскольку файл .mk уже имеет полный контроль над действиями, выполняемыми на каждом шаге построения пакета.

Доступны следующие точкикрепления крючков:

- LIBFOO\_PRE\_DOWNLOAD\_HOOKS
- LIBFOO\_POST\_DOWNLOAD\_HOOKS
- LIBFOO\_PRE\_EXTRACT\_HOOKS
- LIBFOO\_POST\_EXTRACT\_HOOKS
- LIBFOO\_PRE\_RSYNC\_HOOKS
- LIBFOO\_POST\_RSYNC\_HOOKS
- LIBFOO\_PRE\_PATCH\_HOOKS
- LIBFOO\_POST\_PATCH\_HOOKS
- LIBFOO\_PRE\_CONFIGURE\_HOOKS
- LIBFOO\_POST\_CONFIGURE\_HOOKS
- LIBFOO\_PRE\_BUILD\_HOOKS
- LIBFOO\_POST\_BUILD\_HOOKS
- LIBFOO\_PRE\_INSTALL\_HOOKS (только для пакетов x86)
- LIBFOO\_POST\_INSTALL\_HOOKS (только для пакетов x86)
- LIBFOO\_PRE\_INSTALL\_STAGING\_HOOKS (только для целевых пакетов)
- LIBFOO\_POST\_INSTALL\_STAGING\_HOOKS (только для целевых пакетов)
- LIBFOO\_PRE\_INSTALL\_TARGET\_HOOKS (только для целевых пакетов)
- LIBFOO\_POST\_INSTALL\_TARGET\_HOOKS (только для целевых пакетов)
- LIBFOO\_PRE\_INSTALL\_IMAGES\_HOOKS
- LIBFOO\_POST\_INSTALL\_IMAGES\_HOOKS
- LIBFOO\_PRE\_LEGAL\_INFO\_HOOKS
- LIBFOO\_POST\_LEGAL\_INFO\_HOOKS

- LIBFOO\_TARGET\_FINALIZE\_HOOKS

Эти переменные представляют собой список имен переменных, содержащих действия, которые должны быть выполнены в этой точке хука. Это позволяет регулировать несколько хуков в данной точке хука. Вот пример:

```
определить LIBFOO_POST_PATCH_FIXUP действие1
действие2
отпустить

LIBFOO_POST_PATCH_HOOKS += LIBFOO_POST_PATCH_FIXUP
```

### 18.23.1 Использование хука POST\_RSYNC

Хук POST\_RSYNC запускается только для пакетов, которые используют локальный источник, либо через метод локального сайта, либо через механизм OVERRIDE\_SRCDIR. В этом случае исходники пакетов копируются с помощью rsync из локального расположения каталога сборки buildroot. Однако команда rsync не копирует все файлы из исходного каталога. Файлы, принадлежащие системе контроля версий, такие как каталоги .git, .hg и т. д., не копируются. Для большинства пакетов это достаточно, но данный пакет может выполнять дополнительные действия с помощью хука POST\_RSYNC.

В принципе, хук может содержать любую команду, которую вы хотите отдать. Однако одним из конкретных вариантов использования является намеренное копирование каталога управления версиями с помощью rsync. Команда rsync, которую вы используете в хуке, может, среди прочего, использовать следующие переменные:

- \$(SRCDIR): путь к определенному исходному каталогу
- \$(@D): путь к каталогу с борками

### 18.23.2 Хук Target-finalize

Пакеты также могут регулировать хуки в LIBFOO\_TARGET\_FINALIZE\_HOOKS. Эти хуки запускаются после сборки всех пакетов, но до генерации образов файловой системы. Они используются ярдко, ваш пакет, вероятно, не нуждается в них.

## 18.24 Интеграция Gettext и взаимодействие с пакетами

Многие пакеты, поддерживающие интернационализацию, используют библиотеку gettext. Зависимости для этой библиотеки довольно сомнительны и поэтому зачастую включают некоторое ограничение.

Библиотека glibc содержит полную реализацию gettext, поддерживающую перевод. Поэтому поддержка родных языков встроена в glibc.

С другой стороны, библиотеки C и Clibc и musl предоставляют только реализацию -заглушку функции иональности gettext, что позволяет компилировать библиотеки и программы с использованием функции gettext, но не предоставляя возможности перевода полной реализации gettext. С такими библиотеками C, если необходимо использовать поддержку родного языка, ее может обеспечить библиотека libintl пакета gettext.

В связи с этим, а также для того, чтобы убедиться, что поддержка родных языков обрабатывается должным образом, пакеты в Buildroot, которые могут использовать поддержку NLS, должны:

- Убедитесь, что поддержка NLS включена, когда BR2\_SYSTEM\_ENABLE\_NLS=y. Это делается автоматически для пакетов autotools и поэтому должно выполняться только для пакетов, использующих другие инфраструктуры пакетов.
- Добавьте \$(TARGET\_NLS\_DEPENDENCIES) в переменную пакета <pkg>\_DEPENDENCIES. Это добавление должно быть сделано безоговорочно: значение этой переменной автоматически корректируется базовой инфраструктурой для включения соответствующих пакетов. Если поддержка NLS отключена, эта переменная пуста. Если поддержка NLS включена, эта переменная содержит host-gettext, чтобы на нее были доступны инструменты, необходимые для компиляции файлов перевода. Кроме того, если используются языковые пакеты Clibc или musl, эта переменная также содержит gettext для получения полной реализации gettext.

3. При необходимости добавьте `$(TARGET_NLS_LIBS)` к флагам компоновщика, чтобы пакет был с введен с `libintl`. Обычно это не требуется для пакетов `autotools`, поскольку они обычно автоматически определяют, что должны быть с введен с `libintl`. Однако, пакетам, использующим другие системы сборки, или проблемным пакетам на основе `autotools` это может понадобиться `$(TARGET_NLS_LIBS)` должен быть добавлен к флагам компоновщика без каких-либо условий, так как ядро автоматически делает его пустым или определяет как `-lintl` в зависимости от конфигурации.

Для поддержки NLS не следует вносить никаких изменений в файл `Config.in`.

Наконец, некоторым пакетам требуется не некоторые утилиты `gettext` на целевом объекте, такие как сама программа `gettext`, которая позволяет извлекать переведенные строки из командной строки. В таком случае пакет должен:

- использовать `select BR2_PACKAGE_GETTEXT` в своем файле `Config.in`, указав в комментарии выше, что это зависит от времени выполнения только один раз;
- не добавлять никаких зависимостей `gettext` в переменную `DEPENDENCIES` своего файла `.mk`.

## 18.25 Советы и рекомендации

### 18.25.1 Имя пакета, имя записи конфигурации и связь между переменными makefile

В `Buildroot` существует некоторая взаимосвязь между:

- имя пакета, которое представляется общей именем каталога пакета (и имя файла `*.mk`);
- имя записи конфигурации, объявленное в файле `Config.in`;
- префикс переменной `makefile`.

Обязательно необходимо поддерживать согласованность между этими элементами, используя следующие правила:

- каталог пакета и имя `*.mk` представляются общей самой именем пакета (например: `package/foo-bar_boo/foo-bar_boo.mk`);
- имя записи `make` — это само имя пакета (например: `foo-bar_boo`);
- запись конфигурации представляется общей именем пакета заглавными буквами (например: `BR2_PACKAGE_FOO_BAR_BOO`);
- префикс переменной файла `*.mk` — это имя пакета в верхнем регистре (например: `FOO_BAR_BOO_VERSION`).

### 18.25.2 Как проверить стиль кодирования

`Buildroot` предоставляет скрипт в `utils/check-package`, который проверяет новые или измененные файлы на стиль кодирования. Это не полный валидатор языка, но он ловит многие распространенные ошибки. Он предназначен для запуска в реальных файлах, которые вы создали или изменили, перед тем, как создание патча для отправки.

Этот скрипт можно использовать для пакетов, файловых систем `makefiles`, файлов `Config.in` и т. д. Он не проверяет файлы, определяющие пакет, инфраструктуры и некоторые другие файлы, содержащие схожий общий код.

Чтобы использовать его, запустите скрипт проверки пакета, указав, какие файлы вы создали или изменили:

```
$ ./utils/check-package пакет/новый-пакет/*
```

Если в вашем пути есть каталог `utils`, вы также можете запустить:

```
$ cd пакет/новый-пакет/
$ чек-пакет *
```

Инструмент также можно использовать для пакетов в `br2-external`:

```
$ check-package -b /путь/к/br2-ext-tree/пакету/мой-пакет/*
```

Для работы с криптой check-package требуется установить shellcheck и пакеты Python PyPi flake8 и python-magic.

База кода Buildroot в настоящий момент использует версию ShellCheck 0.7.1. Если вы используете другую версию ShellCheck, вы можете увидеть дополнительные, неисправимые предупреждения.

Если у вас есть Docker или Podman, вы можете запустить check-package без установки зависимостей:

```
$ ./utils/docker-run ./utils/check-package
```

### 18.25.3 Как протестировать ваш пакет

После добавления нового пакета важно протестировать его в различных условиях: подходит ли он для всех архитектур?

Собирается ли он с использованием различных библиотек? Нужны ли ему потоки, NPTL? И так далее...

Buildroot запускает **автосборщики**, которые непрерывно проверяют случайные конфигурации. Однако они смотрят только линейную ветку дерева git, а ваш новый модульный пакет еще не там.

Buildroot предоставляет скрипты в utils/test-pkg, который использует те же базовые конфигурации, что и автосборщики, поэтому вы можете протестировать свой пакет в тех же условиях.

Сначала создайте файл конфигурации, который содержит все необходимые параметры, необходимые для включения вашего пакета, но без каких-либо параметров архитектуры или цепочки инструментов. Например, давайте создадим файл конфигурации, который просто включает libcurl, без какого-либо бэкэнда TLS:

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
```

Если вашему пакету нужны дополнительные параметры конфигурации, вы можете добавить их в файл конфигурации. Например, вот как вы можете протестировать libcurl с openssl в качестве TLS-бэкэнда и программой curl:

```
$ cat libcurl.config
BR2_PACKAGE_LIBCURL=y
BR2_PACKAGE_LIBCURL_CURL=y
BR2_PACKAGE_OPENSSL=y
```

Затем запустите скрипт test-pkg, указав ему, какой файл конфигурации использовать и какой пакет тестировать:

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
```

По умолчанию test-pkg будет собираять ваш пакет с использованием подмножества инструментальных цепочек, используемых автосборщиками, которое было выбрано разработчиками Buildroot как наиболее полезное и презентабельное подмножество. Если вы хотите протестировать все инструментальные цепочки, передайте опцию -a. Обратите внимание, что в любом случае внутренние инструментальные цепочки могут быть временно заблокированы из-за занятости.

Выводе перечислены все протестированные цепочки инструментов и соответствующие результаты (флаги, результаты поддельные):

```
$ ./utils/test-pkg -c libcurl.config -p libcurl
      armv5-ctng-linux-gnueabi [ 1/11]: OK armv7-ctng-linux-
      gnueabihf [ 2/11]: OK br-aarch64-glibc [ 3/11]: ПРОПУЩЕНО br-
      arcl-hs38 [ 4/11]: ПРОПУЩЕНО br-arm-basic [ 5/11]:
      НЕ УДАЧНО br-arm-cortex-a9-glibc [ 6/11]: OK
```

```
br-arm-cortex-a9-musl [ 7/11]: НЕ УДАЧНО br-arm-cortex-m4-
full [ 8/11]: OK br-arm-full [ 9/11]: OK br-arm-full-
nothread [10/11]: НЕ УДАЧНО br-arm-
full-static [11/11]: OK
```

11 с борок, 2 пропущено, 2 с борками не удалось, 1 юридическая информация не прошла

Результаты означают:

- OK: сборка прошла успешно.
- ПРОПУЩЕНО: один или несколько параметров конфигурации, перечисленных в фрагменте конфигурации, отсутствовали в окончательной конфигурации. Это означает, что параметрами, имеющими зависимости, не удовлетворяющие цепочкой инструментов, например, пакет, зависящий от BR2\_USE\_MMU с цепочкой инструментов по MMU. Отсутствующие параметры указаны в missing.config в выходном каталоге с сборки (по умолчанию ~ / br-test-pkg / TOOLCHAIN\_NAME /).
- FAILED: сборка не удалась. Проверьте файл журнала в выходном каталоге с сборки, чтобы увидеть, что пошло не так:
  - фактическая сборка не удалась,
  - юридическая информация не

удалась, - один из предварительных шагов (загрузка файла конфигурации, применение конфигурации, запуск dirclean для пакета) неуспешный.

При возникновении ошибки вы можете просто повторно запустить скрипт с теми же параметрами (после исправления пакета); скрипт попытается пересобрать пакет, указанный с помощью -р для всех цепочек инструментов, без необходимости пересобирать все зависимости этого пакета.

Скрипт test-pkg принимает несколько параметров, с правкой которых можно получить, выполнив:

```
$ ./utils/test-pkg -h
```

#### 18.25.4 Как добавить пакет из GitHub

Пакеты на GitHub часто не имеют области загрузки с релизовыми tarballs. Однако можно загрузить tarballs напрямую из репозитория на GitHub. Поскольку известно, что GitHub в прошлом менял механизмы загрузки, следует использовать вспомогательную функцию github, как показано ниже.

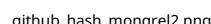
```
# Используйте тег или полный идентификатор
коммита FOO_VERSION = 1.0
FOO_SITE = $(call github,<user>,<package>,v$(FOO_VERSION))
```

##### ПРИМЕЧАНИЯ

- FOO\_VERSION может быть либо тегом, либо идентификатором фиксации.
- Имя tarball, сгенерированное github, совпадает с именем по умолчанию из Buildroot (например: foo-f6fb6654af62045239caed5950bc6c), поэтому нет необходимости указывать его в файле .mk.
- При использовании идентификатора фиксации в качестве версии следует использовать полные 40-байтные хэш-именования.
- Если тег содержит префикс, такой как v1.0, то переменная VERSION должна содержать только 1.0, а не должна быть добавлена непосредственно в переменную SITE, как показано выше. Это гарантирует, что значение переменной VERSION может быть использовано для отслеживания [release-monitoring.org](https://release-monitoring.org) результаты.

Если пакет, который вы хотите добавить, имеет раздел релиза на GitHub, то сопровождающий его загрузка релизный tarball, или релиз может просто указываться на автоматически сгенерированный tarball из тега git. Если есть релизный tarball, загруженный сопровождающим, мы предпочитаем использовать его, поскольку он может немногим отличаться (например, он содержит скрипты конфигурации, поэтому нам не нужно выполнять AUTORECONF).

На странице релиза можно увидеть, загружен ли tarball или git-tag:



- Если он выглядит как на изображении выше, то он был загружен сопровождающим, и вам следует использовать эту ссылку (в этом примере: mongrel2-v1.9.2.tar.bz2) для указания FOO\_SITE, а не использовать помощник github.
- С другой стороны, если есть только ссылка «Исходный код», то это автоматически сгенерированный tarball, и вам следует использовать вспомогательную функцию github.

## 18.25.5 Как добавить пакет из Gitlab

Аналогично макросу `github`, описанному в разделе [18.25.4](#), `Buildroot` также предоставляет макрос `gitlab` для загрузки из репозиториев `Gitlab`. Его можно использовать для загрузки автоматически сгенерированных `tarballs`, созданных `Gitlab`, либо для определенных тегов, либо для коммитов:

```
# Используйте тег или полный идентификатор  
коммита FOO_VERSION = 1.0  
FOO_SITE = $(call gitlab,<user>,<package>,v$(FOO_VERSION))
```

По умолчанию будет использоваться `tar`-архив `.tar.gz`, но `Gitlab` также предоставляет `tar`-архивы `.tar.bz2`, поэтому, добавив переменную `<pkg>_SOURCE`, можно использовать этот `tar`-архив `.tar.bz2`:

```
# Используйте тег или полный идентификатор  
коммита FOO_VERSION = 1.0  
FOO_SITE = $(call gitlab,<user>,<package>,v$(FOO_VERSION))  
FOO_SOURCE = foo-$(FOO_VERSION).tar.bz2
```

Если разработчик использует новую ветку разработки и загрузил определенный `tarball` по адресу <https://gitlab.com/<project>/releases/>, не используйте этот макрос, а используйте прямую ссылку на `tarball`.

## 18.26 Заключение

Как видите, добавление программного пакета в `Buildroot` сводится к просмотру написания `Makefile` с использованием существующего примера и изменению его в соответствии с процессом компиляции, требуемым пакетом.

Если вы упаковываете программное обеспечение, которое может быть полезно другим людям, не забудьте отправить патч в список рассылок `Buildroot` (см. раздел [22.5](#))!

# Глава 19

## Исправление пакета

При интеграции нового пакета или обновлении существующего может потребоваться исправление исходного кода программы обес печения для его кросс-сборки в Buildroot.

Buildroot предлагает инфраструктуру для автоматической обработки этого во время сборки. Он поддерживает три способа применения наборов патчей: загруженные патчи, патчи, поставляемые в buildroot, и патчи, расположенные в определяемом пользователем глобальном каталоге патчей.

### 19.1 Предоставление исправлений

#### 19.1.1 Скачано

Если необходимо применить патч, доступный для загрузки, добавьте его в переменную <packagename>\_PATCH. Если запись содержит://, то Buildroot предположит, что это полный URL, из которого будет загружено патч из этого места. В противном случае Buildroot предположит, что патч следует загрузить с <packagename>\_SITE. Это может быть отдельный патч или tarball, содержащий серию патчей.

Как и для всех загрузок, в файл <packagename>.hash необходимо добавить хеш.

Этот метод обычно используется для пакетов Debian.

#### 19.1.2 Внутри Buildroot

Большинство исправлений предоставляются в Buildroot, в каталоге пакета; они обычно направлены на исправление кросскомпиляции, поддержки libcs или других подобных проблем.

Эти файлы исправлений должны называться <номер>-<описание>.patch.

#### ПРИМЕЧАНИЯ

- Файлы исправлений, поставляемые в Buildroot, не должны содержать в имени файла никаких ссылок на версию пакета.
- Поле <номер> в имени файла исправления относится к порядку применения и должно начинаться с 1; предпочтительно дополнить номер нули до 4 цифр, как это делает git-format-patch. Например: 0001-foobar-the-buz.patch
- Префикс темы листа с патчем не должен быть пронумерован. Патчи должны быть сгенерированы с помощью команды git format-patch -N, поскольку эта нумерация автоматически добавляется к серии. Например, строка темы патча должна выглядеть как Subject: [PATCH] foobar the buz, а не Subject: [PATCH n/m] foobar the buz.
- Раньше для патчей было обязательным префикс в виде имени пакета, например <package>-<number>-<description>, но теперь это не так. Существующие пакеты будут исправлены с временем. Не добавляйте к патчам префикс в виде имени пакета.
- Ранее файл с серией, используемый quilt, также мог быть добавлен в каталог пакета. В этом случае файл с серией определяет порядок применения патчей. Это устарело и будет удалено в будущем. Не используйте файл с серией.

### 19.1.3 Глобальный каталог исправлений

Параметр файла конфигурации BR2\_GLOBAL\_PATCH\_DIR может использовать я для указания списка из одного или нескольких каталогов, разделенных пробелами, содержащих глобальные исправления пакетов. Подробности см. в разделе [9.8.1](#).

## 19.2 Как применять патчи

1. Запустите команды <packagename>\_PRE\_PATCH\_HOOKS, если они определены;
2. Очистите каталог с борками, удалив все существующие файлы \*.rej;
3. Если определено <packagename>\_PATCH, то применяются исправления из этих tarballs;
4. Если в каталоге Buildroot пакета или в подкаталоге пакета с именем <packageversion> есть файлы \*.patch, затем:
  - Если файл серии существует в каталоге пакета, то патчи применяются в соответствии с файлом серии; • В противном случае файлы патчей, соответствующие \*.patch, применяются в алфавитном порядке. Поэтому, чтобы гарантировать их применение в правильном порядке, настоятельно рекомендуется называть файлы патчей следующим образом: <номер>-<описание>.patch, где <номер> относится к порядку применения.
5. Если определен BR2\_GLOBAL\_PATCH\_DIR, каталоги будут перечислены в том порядке, в котором они указаны. Патчи применяются как описано в предыдущем шаге.
6. Запустите команды <packagename>\_POST\_PATCH\_HOOKS, если они определены.

Если на шагах 3 или 4 что-то пойдет не так, с борком завершится неудачей.

## 19.3 Формат и лицензирование пакетов исправлений

Патчи выпускаются под той же лицензией, что и программное обеспечение, к которому они применяются (см. раздел [13.2](#)).

В заголовке комментария к патчу следует добавить сообщение, объясняющее, что делает патч и почему он необязателен.

Вам следует добавить заявление о подписании в заголовок каждого исправления, чтобы помочь отслеживать изменения и подтверждать, что исправление выпущено под той же лицензией, что и измененное программное обеспечение.

Если программное обеспечение находится под контролем версий, рекомендуется использовать программное обеспечение SCM верхнего уровня для генерации набора исправлений.

В противном случае объедините заголовок с выводом команды diff -purN package-version.orig/ package-version/.

Если вы обновляете существующий патч (например, при повышении версии пакета), убедитесь, что существующие заголовки From и тегги Signed-off-by не удалены, но при необходимости обновите остальную часть комментария к патчу.

В итоге патч должен выглядеть так:

```
configure.ac: добавить тест поддержки C++
Подпись: Джон Дью <john.doe@noname.org>
--- настроить.ac.orig +++
настройка.ac @@ -40,2
+40,12 @@
AC_PROG_MAKE_SET
+
+AC_CACHE_CHECK([работает ли компилятор C++],
+ [rw_cv_prog_cxx_works],
```

```
+          [AC_LANG_PUSH([C++)]
+          AC_LINK_IFELSE([AC_LANG_PROGRAM([], [])],
+                         [rw_cv_prog_cxx_works=да],
+                         [rw_cv_prog_cxx_works=нет])
+          AC_LANG_POP([C++]))
+
+AM_CONDITIONAL([CXX_WORKS], [тест "x$rw_cv_prog_cxx_works" = "xyes"])
```

## 19.4 Дополнительная документация по исправлению

В идеале все исправления должны документировать исправление в исходной версии или отправку исправления если это применимо, через трейлер в исходной версии.

При обратном портировании патча из основной ветки, который был принят в основную ветку, желательно указать URL-адрес коммита:

Upstream: <URL для upstream-коммита>

Если в Buildroot обнаружена новая проблема, и она затрагивает всю вышеуказанную ветку (это не проблема, связанные с Buildroot), пользователи следуют отправить исправление в вышеуказанную ветку и предоставить ссылку на эту отправку, когда это возможно:

Upstream: <URL для отправки списка ссылок upstream или запроса слияния>

Патчи, которые были отправлены, но отклонены вышеуказанный инстанцией, должны иметь пометку об этом и включать комментарии о том, почему был отправлен патч. Используется несогласие с тегом вышеуказанного организатора.

Примечание: в любом из вышеуказанных случаев также разумно добавить несколько слов о любых изменениях в патче, которые могли быть необязательными.

Если исправление не применяется к вышеуказанной версии, то это следует отметить в комментарии:

Upstream: N/A <дополнительная информация о том, почему патч не применичен для Buildroot>

Добавление этой документации помогает оптимизировать процесс проверки исправлений во время обновлений версий пакетов.

# Глава 20

## Инфраструктура загрузки

ВСЕ

## Глава 21

# Отладка Buildroot

Можно инструментировать шаги, которые Buildroot делает при сборке пакетов. Определите переменную `BR2_INSTRUMENTATION_SCRIPTS`, чтобы она содержала путь к одному или нескольким скриптам (или другим полняемым файлам) в списке, разделенном пробелами, которые вы хотите вызывать до и после каждого шага. Скрипты вызываются я последовательно с тремя параметрами:

- начало или конец для обозначения начала (соответственно конца) шага;
- название шага, который должен начаться или который только что завершился;
- название пакета.

Например:

```
сделать BR2_INSTRUMENTATION_SCRIPTS="/путь/к/моему/скрипту1 /путь/к/моему/скрипту2"
```

Список шагов:

- извлекать
- плести
- настроить
- строить
- install-host, когда пакет установлен в `$(HOST_DIR)`
- install-target, когда целевой пакет установлен в `$(TARGET_DIR)`
- install-staging, когда целевой пакет устанавливается в `$(STAGING_DIR)`
- install-image, когда целевой пакет устанавливает файлы в `$(BINARIES_DIR)`

Скрипт имеет доступ к следующим переменным:

- `BR2_CONFIG`: путь к файлу `Buildroot.config`
- `HOST_DIR`, `STAGING_DIR`, `TARGET_DIR`: см. раздел [18.6.2](#)
- `BUILD_DIR`: каталог, в который извлекаются и собираются пакеты
- `BINARIES_DIR`: место, где находятся двоичные файлы (они же изображения)
- `BASE_DIR`: базовый выходной каталог
- `PARALLEL_JOBS`: количество заданий, используемых при запуске параллельных процессов.

## Глава 22

# Вклад в Buildroot

Есть много способов, которыми вы можете внести свой вклад в Buildroot: анализ и исправление ошибок, анализ и исправление сбоев в борки пакетов, обнаруженных автоборщиками, тестирование и рассмотрение исправлений, отправленных другими разработчиками, работа над элементами в нашем списке TODO и отправка соответственных улучшений в Buildroot или его руководство. В следующих разделах немног о подробнее о каждом из этих элементов.

Если вы заинтересованы в однажды в Buildroot, первое, что вам следует делать, это подписать яна список патчей Buildroot. Этот список является основным способом взаимодействия с другими разработчиками Buildroot и отправки вкладов. Если вы еще не подписаны, то обратитесь к Главе 5 для просмотра списка подписки.

Если вы собираетесь работать с кодом, настоятельно рекомендуется использовать git-репозиторий Buildroot, а не начинать с извлечения исходного кода tarball. Git — это самый простой способ разработки и прямой отправки ваших патчей в список патчей. Обратитесь к Главе 3 для получения дополнительной информации о получении git-директории Buildroot.

### 22.1 Воспроизведение, анализ и исправление ошибок

Первый способ внести свой вклад — просмотреть открытые отчеты об ошибках в системе отслеживания ошибок Buildroot. Поскольку мы стремимся внести количества ошибок к минимуму, любая помощь в воспроизведении, анализе и исправлении сообщенных ошибок будет более чем приветствоваться. Не стесняйтесь добавлять комментарии к отчетам об ошибках, сообщая о своих находках, даже если вы еще не видите полной картины.

### 22.2 Анализ и исправление сбоев в автоборки

Автоборщик Buildroot предоставляет собой набор сборочных машин, которые непрерывно запускаются с борками Buildroot на основе случайных конфигураций. Это делается для всех архитектур, поддерживаемых Buildroot, с различными цепочками инструментов и с лучшим выбором пакетов. С большой активностью коммитов на Buildroot эти автоборщики являются яблочной помощью в обнаружении проблем на очень ранних стадиях после коммита.

Все результаты сборки доступны на <http://autobuild.buildroot.org>. Статистика доступна по адресу <http://autobuild.buildroot.org/stats.php>. Каждый день в список патчей ссылаются обзоры всех невыполненных пакетов.

Обнаружение проблем — это здорово, но очевидно, что эти проблемы также должны быть устранены. Ваш вклад здесь очень приветствуется! Помимо этого, можно делать две вещи:

- Анализ проблем. Ежедневные водяные письма не содержат подробностей о фактических сбоях: чтобы увидеть, что происходит, нужно открыть журнал с борками и проверить последний вывод. Наличие кого-то, кто делает это для всех пакетов в почте, очень полезно для других разработчиков, поскольку они могут сделать быстрый начальный анализ, основываясь только на этом выводе.
- Устранение проблемы. При устранении сбоев автоборки следуют выполнить следующие действия:
  1. Проверьте, можете ли вы воспроизвести проблему, сбросив ее с той же конфигурацией. Вы можете сделать это вручную или использовать `br-reproduce-build` скрипта, который автоматически клонирует git-репозиторий Buildroot, извлекает правильную ревизию, загружает и устанавливает правильную конфигурацию, а затем запускает с боркой.

2. Проанализируйте проблему и найдите решение.
3. Убедитесь, что проблема действительностранена, начав с чистого дерева Buildroot и применив только свое исправление.
4. Отправьте исправление в список рассылки Buildroot (см. раздел [22.5](#)). В случае, если вы создали патч для нескольких пакетов, вам также следует отправить патч в апстрим, чтобы проблема была исправлена в более позднем релизе, а патч в Buildroot можно было удалить. В сообщении о фиксации патча, исправляющего ошибку автосборки, добавьте ссылку на каталог результатов сборки следующим образом:

Исправления <http://autobuild.buildroot.org/results/51000a9d4656afe9e0ea6f07b9f8ed374c2e4069>

## 22.3 Проверка и тестирование исправлений

С таким количеством патчей, отправляемых в список рассылки каждый день, управляющие очень сложной работой оценки того, какие патчи готовы к применению, а какие нет. Участники могут оказать здесь большую помощь, просматривая и тестируя эти патчи.

В процессе обзора не стесняйтесь отвечать на исправления для замечаний, предложений или чего-либо, что поможет всем понять исправления и сделать их лучше. Пожалуйста, используйте интернет-стиль ответов в простых текстовых письмах при ответе на заявки на исправления.

Для обозначения одобрения патча есть три формальных тега, которые отслеживают это одобрение. Чтобы добавить свой тег к патчу, отметьте на него отегом одобрения под строкой Signed-off-by каждого автора. Эти теги будут автоматически подписаны patchwork (см. раздел [22.3.1](#)) и станут частью журнала коммитов, когда патч будет принят.

### Tested-by

Указывает, что патч был успешно протестирован. Вам предлагается указать, какой тип тестирования вы провели (компиляционный тест на архитектуре X и Y, тест времени выполнения на A, ...). Эта дополнительная информация помогает другим тестировщикам сопровождающему.

### Проверено

Указывает, что вы проверили код патча и сделали все возможное для обнаружения проблем, но вы недостаточно знакомы с затронутой областью, чтобы предстать перед тегом Acked-by. Это означает, что в патче могут остаться проблемы, которые будут обнаружены кем-то с большим опытом в этой области. Если такие проблемы будут обнаружены, ваш тег Reviewed-by станет якобыенным, и вас нельзя будет обвинить.

### Acked-by

Указывает, что вы проверили код патча и достаточно знакомы с затронутой областью, чтобы чувствовать, что патч может быть зафиксирован как есть (дополнительные изменения не требуются). Если позже выясняется что с патчем что-то не так, ваш Acked-by может быть считан неподходящим. Разница между Acked-by и Reviewed-by заключается в том, что вы готовы взять на себя вину за патч Acked, но не за Reviewed.

Если вы рассматриваете патч и у вас есть комментарии по нему, вам следует просто ответить на патч, указав эти комментарии, не предстavляя тег Reviewed-by или Acked-by. Этот тег и следует предстavлять только в том случае, если вы считаете, что патч как таковой.

Важно отметить, что ни Reviewed-by, ни Acked-by не подразумевают, что тестирование было проведено. Чтобы указать, что вы оба просмотрели и протестировали патч, укажите два отдельных тега (Reviewed/Acked-by и Tested-by).

Обратите внимание также, что любой разработчик может предстать перед тегом Tested/Reviewed/Acked-by без исключения и мы призываем всех делать это. Buildroot не имеет определенной группы новых разработчиков, просто так получается, что некоторые разработчики более активны, чем другие. Сопровождающий будет оценивать тег и в соответствии с должностным списком отправителя. Теги, представленные постым участником, если тестенно, будут пользоваться большим доверием, чем теги, представленные новичком. Помимо этого, как вы представляете теги более регулярно, ваша надежность (в глахах сопровождающим) будет расти, но любой представленный тег ценен.

Веб-сайт Buildroot's Patchwork может использовать языки патчей в целях тестирования. Более подробную информацию об использовании веб-сайта Buildroot's Patchwork для применения патчей см. в разделе [22.3.1](#).

### 22.3.1 Применение лос кутног о шитья

Основное применение веб-сайта Buildroot Patchwork для разработчиков — загрузка патчей в локальный репозиторий git для целей тестирования.

При просмотре патчей в интерфейсе управления патчворком вверху страницы отображается ссылка mbox. Скопируйте адрес этой ссылки и выполните следующие команды:

```
$ git checkout -b <имяветки-теста> $ wget -O - <url-mbox> | git am
```

Другой вариант применения патчей — создание пакета. Пакет — это набор патчей, которые можно сгруппировать с помощью интерфейса patchwork. После создания пакета и публикации пакеты можете скопировать ссылку mbox на пакет и применить пакет с помощью приведенных выше команд.

### 22.4 Работа над пунктами из списка TODO

Если вы хотите внести свой вклад в Buildroot, но не знаете, с чего начать, и вам не нравятся ни одна из выше перечисленных тем, вы все еще можете поработать над пунктами из [списка TODO Buildroot](#). Не стесняйтесь сначала обсудить элемент в списке рассылки или на IRC. Редактируйте вики, чтобы указать, когда вы начинаете работать над элементом, чтобы мы избегали дублирования или.

### 22.5 Отправка ис правлений

---

Примечание. Пожалуйста, не прикрепляйте ис правления к ошибкам, вместе с этими отправляйте их в список рассылки.

---

Если вы внесли какие-либо изменения в Buildroot и хотите внести их в проект Buildroot, выполните следующие действия.

#### 22.5.1 Форматирование патча

Мы ожидаем, что патчи будут отформатированы определенным образом. Это необходимо для того, чтобы было легкое просматривать патчи, чтобы можно было легко применять их к репозиторию git, чтобы было легко найти историю, как и почему что-то изменилось, и чтобы можно было использовать git bisect для определения ис точника проблемы.

Прежде всего, важно, чтобы патч имел короткое сообщение о коммите. Сообщение о коммите должно начинаться с отдельной строки с кратким описанием изменения с префиксом области, затронутой патчем. Несколько примеров коротких названий коммитов:

- package/linuxptp: повысить версию до 2.0
- configs/imx23evk: повышение версии Linux до 4.19
- package/pkg-generic: отложить аренду условий зависимости
- boot/u-boot: требуется ядро host-{flex,bison}
- поддержка/тестирование: добавление тестов python-ubjson

Описание, следующее за префиксом, должно начинаться с оточной буквы (например, «bump», «needs», «postpone», «add» в приведенных выше примерах).

Во вторых, текст сообщения о коммите должен описывать, почему это изменение необходимо, и при необходимости также давать подробности о том, как это было сделано. При написании сообщения о коммите подумайте о том, как его прочтут рецензенты, но также подумайте о том, как вы прочтете его, когда снова посмотрите на это изменение через несколько лет.

В-третьих, сам патч должен вносить только одно изменение, но делать его полностью. Две не связанные или с лабо связанные изменения обычно следует вносить в два отдельных патча. Обычно это означает, что патч влияет только на один пакет. Если не сколько изменений связаны, часто все еще возможно разделить их на небольшие патчи и применить их в определенном порядке. Небольшие патчи облегчают просмотр и часто облегчают поиск ледяной погоды тога, почему было сделано изменение. Однако каждый патч должен быть полным. Не допускается, чтобы с борка была с ломана, когда применяется я только первый, но не второй патч. Это необходимо для возможности использования git bisect в поиске леди.

Конечно, покажи занимается разработкой, вы, вероятно, перемещаетесь между пакетами и, конечно, не фиксируете все с разу, чтобы это было дос таточно чисто для отправки. Поэтому большинство разработчиков переписывают историю коммитов, чтобы создать чистый набор коммитов, подходящий для отправки. Для этого вам нужно использовать интерактивное перебазирование. Вы можете узнать об этом в книге Pro Git. И тогда еще проще сбросить историю с помощью git reset --soft origin/master и выбрать отдельные изменения с помощью git add -i или git add -p.

Наконец, патч должен быть подписан. Это делается путем добавления Signed-off-by: Ваше настоящее имя <your@email.address> в конце сообщения о коммите. git commit -s делает это за вас, если настроено правильно. Тег Signed-off-by означает, что вы публикуете патч под лицом компании Buildroot (т. е. GPL-2.0+, за исключением патчей пакетов, которые имеют лицом upstream), и что вам разрешено это делать. См. [Developer Certificate of Origin](#) для получения подробной информации.

Чтобы указать сопсона с создания патча или процесса просмотра, вы можете использовать поддадресацию электронной почты, для вашей идентификации git (т. е. тога, что используется в качестве поля «Автор коммита» и email From; а также вашего тега Signed-off-by); добавьте с уффиксом к локальной части, отделив ее от знака «плюс +». Например:

- для компании, которая сопсировала предстартовую работу, используйте название компании в качестве поддадресации (с уффиксом) части:

Ваше-Имя Ваша-Фамилия<ваше-имя ваша-фамилия+название-компании@mail.com>

- для лица, сопсировавшего предстартовую работу, используйте его имя и фамилию:

Ваше-Имя Ваша-Фамилия<ваше-имя ваша-фамилия+их-имяих-фамилия@mail.com>

В качестве альтернативы, особенно если ваш почтовый сервер не поддерживает поддадресацию, вы можете включить сопсона в имя автора в скобках, например, «Ваше имя(Имя сопсона)».

При добавлении новых пакетов следует отправлять каждый пакет в отдельном патче. Этот патч должен содержать обновление package/Config.in, файл Config.in пакета, файл .mk, файл .hash, любой скрипт init и все патчи пакета. Если у пакета много подопечных, их иногда лучше добавлять как отдельные посты ледяные патчи. Строки резюме должны быть примерно такими: <packagename>: new package. Тело сообщения о коммите может быть пустым для постых пакетов или может содержать описание пакета (например, текст с правками Config.in). Если для пакета необходи мос делать что-то особенное, это также должно быть явно объяснено в теле сообщения о коммите.

Когда вы переводите пакет на новую версию, вам также следует отправить отдельный патч для каждого пакета. Не забудьте обновить файл .hash или добавить его, если он еще не существует. Также не забудьте проверить, действительно ли \_LICENSE и \_LICENSE\_FILES. Строки с водкой должна быть примерно такой: <packagename>: bump to version <new version>. Если новая версия содержит только обновления безопасности по сравнению с существующей, строка должна быть такой: <packagename>: security bump to version <new version>, в теле сообщения о коммите должны быть указаны номера CVE, которые исправлены. Если некоторые исправления пакета могут быть удалены в новой версии, следуют явно объяснять, почему их можно удалить, желательно с идентификатором файла или в их одной ветке. Также следует явно объяснять любые другие требуемые изменения, например, параметры конфигурации, которые больше не существуют или больше не нужны.

Если вы заинтересованы в получении уведомлений об ошибках с борки и дальнейших изменениях в добавленных или измененных вами пакетах, добавьте себя в файл DEVELOPERS. Это должно быть сделано в том же патче, в котором создается или изменяется пакет. Для получения дополнительной информации см. файл [DEVELOPERS](#).

Buildroot предоставляет удобный инструмент для проверки рас пространенных ошибок с помощью кодирования в созданных или измененных вами файлах, называемый check-package (для получения дополнительной информации см. раздел [18.25.2](#)).

## 22.5.2 Подготовка серии патчей

Начиная с изменений, зафиксированных в локальном предстартовании git, переместите ветку разработки на вершину дерева upstream перед генерацией набора патчей. Для этого выполните:

```
$ git fetch --all --tags $ git rebase
origin/master
```

Теперь проверьте стиль кодирования на предмет внесенных вами изменений:

```
$ utils/docker-run make check-package
```

Теперь вы готовы сгенерировать и отправить свой набор исправлений.

Чтобы сгенерировать его, выполните:

```
$ git format-patch -M -n -s -o исходный origin/master
```

Это приведет к созданию файлов исправлений в исходном подкаталоге, автоматически добавляя строку Signed-off-by.

После создания файлов исправлений вы можете просмотреть/редактировать сообщение о коммите перед их отправкой, используя ваш любимый текстовый редактор.

Buildroot предоставляет удобный инструмент, позволяющий узнать, кому следует отправлять ваши патчи, называемый get-developers (см. Главу 23 для получения дополнительной информации). Этот инструмент считывает ваши патчи и выводит соответствующую команду git send-email для использования

```
$ ./utils/get-developers исходный/*
```

Используйте вывод get-developers для отправки своих патчей:

```
$ git send-email --to buildroot@buildroot.org --cc bob --cc alice исходный/*
```

В качестве альтернативы get-developers -e можно использовать напрямую с аргументом --cc-cmd для git send-email, чтобы автоматически отправить копию затронутым разработчикам:

```
$ git send-email --to buildroot@buildroot.org \
--cc-cmd './utils/get-developers -e' origin/master
```

git можно настроить на автоматическое выполнение этого с разумом из коробки с помощью:

```
$ git config sendemail.to buildroot@buildroot.org $ git config
sendemail.ccCmd "$(pwd)/utils/get-developers -e"
```

А затем просто сделайте:

```
$ git send-email origin/master
```

Обратите внимание, что git должен быть настроен на использование вашего почтового аккаунта. Чтобы настроить git, см. man git-send-email или <https://git-send-email.io/>.

Если вы не используете git send-email, убедитесь, что публикуемые патчи не переносятся на новую строку, иначе их будет сложно применить.

В таком случае исправьте свой почтовый клиент или, что еще лучше, научитесь использовать git send-email.

<https://sr.ht> также имеет легкий пользовательский интерфейс для [подготовки серии патчей](#) и также может отправлять вам патчи. У этого есть несолько недостатков, так как вы не можете редактировать статус ваших патчей в Patchwork и в настоящем времени не можете редактировать свое отображаемое имя с которым отправляются письма с овпадением, но это вариант, если вы не можете заставить git send-email работать с вашим почтовым провайдером (т. е.

О365); это не следует считать официальным способом отправки исправлений, а лишь запасным вариантом.

### 22.5.3 Сопроводительное письмо

Если вы хотите представить весь набор патчей в отдельном письме, добавьте --cover-letter к команде git format-patch (см. man git-format-patch для получения дополнительной информации). Это сгенерирует шаблон для вводного описания вашей серии патчей.

Сопроводительное письмо может быть полезно для представления предлагаемых вами изменений в следующих случаях:

- большое количество коммитов в серии;

- глубокое влияние изменений на остальную часть проекта;
- запрос предложений<sup>1</sup>
- всейкий раз, когда вы чувствуете, что это поможет вам предстать с вашим выбором, процесс рецензирования и т. д.

#### 22.5.4 Патчи для веток обслуживания

При исправлении ошибок в ветке обслуживания сначала следует исправить ошибки в ветке master. Журнал коммита для такого патча может затем содержать примечание после коммита, указывающее, какие ветви затронуты:

```
package/foo: исправить некоторые вещи
```

Подпись анонса: Ваше настоящее имя <your@email.address>

Обратное портирование на: 2020.02.x, 2020.05.x

(2020.08.x не затронута, так как версия была повышена)

Затем эти изменения будут перенесены ответственным за поддержку в затронутые ветви.

Однако некоторые ошибки могут относиться только к определенному релизу, например, потому что они используют более старую версию пакета. В этом случае патчи должны основываться на ветке обслуживания, а префикс темы патча должен включать имя ветви обслуживания (например, "[PATCH 2020.02.x]"). Это можно делать с помощью флага git format-patch --subject-prefix:

```
$ git format-patch --subject-prefix "PATCH 2020.02.x" \
-M -s -o исходный origin/2020.02.x
```

Затем отправьте патчи с помощью git send-email, как описано выше.

#### 22.5.5 Журнал изменений патча

Когда запрашиваются улучшения новая редакция каждого коммита должна включать список изменений между каждым представлением. Обратите внимание, что когда ваша серия патчей представлена сопроводительным письмом, в сопроводительное письмо может быть добавлен общий список изменений в дополнение к списку изменений в отдельных коммитах. Лучший способ переработать серию патчей — это интерактивное перебазирование: git rebase -i origin/master. Задокументированной информации обратитесь к руководству git.

При добавлении к отдельным коммитам этого списка изменений добавляется при редактировании сообщения коммита. Ниже раздела Signed-off-by добавьте --- и ваш список изменений.

Хотя список изменений будет виден рецензентам в почтовой ветке, а также в patchwork, git автоматически пронумерует строки ниже --- когда патч будет объединен. Это предполагаемое поведение: журнал изменений не предназначен для того, чтобы сюда рангом добавлять историю git проекта.

Далее рекомендуемая форма:

Название патча: краткое описание, максимум 72 символов

Абзац, который объясняет проблему и то, как она проявляется. Если проблема ложная, можно добавить больше абзацев. Всегда абзацы должны быть разбиты на 72 символа.

Абзац, который объясняет основную причину проблемы. Отдельно же, больше одного абзаца — это нормально.

Наконец, один или несколько абзацев, объясняющих, как решается проблема. Не стесняйтесь подробно объяснять ложные решения.

Подпись анонса: Джон ДОУ <john.doe@example.net>

<sup>1</sup>RFC: (Запрос комментариев) предложение по изменению

---

Изменения v2 -> v3: - foo bar  
 (предложено Jane) - bar buz

Изменения v1 -> v2:

- альфа браво (предложено Джоном) - чарли дельта

Любая редакция патча должна включать номер версии. Номер версии просто состоит из буквы v, за которой следует целое число, большее или равное двум (т. е. "PATCH v2", "PATCH v3" ...).

Это можно легко сделать с помощью git format-patch, используя опцию --subject-prefix:

```
$ git format-patch --subject-prefix "PATCH v4" \
-M -s -o исходный источник/главный
```

Начиная с версии git 1.8.1, вы также можете использовать -v <n> (где <n> — номер версии):

```
$ git format-patch -v4 -M -s -o исходный origin/master
```

При представлении новой версии патча, пожалуйста, отметьте старую версию как замененную в [патче](#). Вам необязательно создать учетную запись на [patchwork](#), чтобы иметь возможность изменять статус ваших патчей. Обратите внимание, что вы можете изменять статус только тех патчей, которые вы отправили сами, что означает адрес электронной почты, который вы зарегистрировали в [patchwork](#). Патч должен сопровождаться тем, который вы используете для отправки и исправлений в список рассылок.

Вы также можете добавить опцию --in-reply-to=<message-id> при отправке патчей в список рассылок. Идентификатор сообщения, на которое нужно ответить, можно найти под тегом "Message Id" на [patchwork](#). Преимущество функции in-reply-to заключается в том, что patchwork автоматически помечает предыдущую версию патча как замененную.

## 22.6 Собщение о проблемах /ошибках или получение помощи

Прежде чем сообщать о какой-либо проблеме, проверьте [архив с сообщениями](#), возможно, кто-то уже сообщал о подобной проблеме и/или исправил ее.

Независимо от того, какой способ сообщения об ошибках или получения помощи вы выберете — открыв сообщение об ошибке в [системе отслеживания ошибок](#) или отправив сообщение в список рассылок — вам необязательно представлять ряд данных, чтобы помочь другим воспроизвести ошибку и найти решение.

Попробуйте подумать так, как будто вы пытаетесь помочь кому-то другому. Чтобы вам понадобилось в таком случае?

Вот краткий список сведений, которые необходимо предоставить в таком случае:

- ОС/машина (ОС/выпуск)
- версия Buildroot
- цель, для которой сборка не удалась
- пакет(ы), для которых сборка не удалась
- команда, которая не выполняется и ее вывод
- любая информация, которая по вашему мнению, может быть важна

Кроме того, вам следует добавить файл .config (или, если вы знаете как, defconfig; см. раздел [9.3](#)).

Если некоторые из этих деталей слишком велики, не стесняйтесь использовать сервис pastebin. Обратите внимание, что не все доступные сервисы pastebin сокращают разделятели строк в стиле Unix при загрузке сырых текстов. Известно, что следующие сервисы pastebin работают правильно: <https://gist.github.com/> - <http://code.bulix.org/>

## 22.7 Использование фреймворка тестов времени выполнения

Buildroot включает в себя фреймворк тестирования во время выполнения, построенный на языке Python и выполнении во время выполнения QEMU. Цели фреймворка следующие:

- создать четко определенную конфигурацию Buildroot
- при желании проверьте некоторые свойства одних данных с борки
- при желании загрузите результаты с борки в Qemu и проверьте, что заданная функция работает так, как ожидалось

Точкой входа для использования с реды тестирования времени выполнения является инструмент support/testing/run-tests, который имеет ряд параметров, задокументированных в описании help -h инструмента. Некоторые общие параметры включают установку папки загрузки, вых однапапки, с охранение вых одных данных с борки, а для нескольких тестовых случаев вы можете установить JLEVEL для каждого.

Ниже приведен пример шагов о выполнения тестового случая.

- Для начала давайте посмотрим, какие есть все варианты тестовых случаев. Тестовые случаи можно перечислить, выполнив support/testing/run-test -l. Вс эти тесты можно запускать по отдельности во время разработки тестов с консоли. Как по одному, так и выборочно как группу подмножество тестов.

```
$ поддержка/тестирование/запуск -l
Список тестов
test_run (tests.utils.test_check_package.TestCheckPackage) test_run
(tests.toolchain.test_external.TestExternalToolchainBuildrootMusl) ... ок test_run (tests.toolchain.test_external.TestExternalToolchainBuildrootuClibc) ...
ок test_run (tests.toolchain.test_external.TestExternalToolchainCCache) ... ок test_run (tests.toolchain.test_external.TestExternalToolchainCtngMusl) ...
ок test_run (tests.toolchain.test_external.TestExternalToolchainLinaroArm) ... ок test_run
(tests.toolchain.test_external.TestExternalToolchainSourceryArmv4) ... ок test_run
(tests.toolchain.test_external.TestExternalToolchainSourceryArmv5) ... ок тестовый_запуск
(tests.toolchain.test_external.TestExternalToolchainSourceryArmv7) ... ок [snip] тестовый_запуск
(tests.init.test_systemd.TestInitSystemSystemdRoFull) ... ок тестовый_запуск (tests.init.test_systemd.TestInitSystemSystemdRoIfupdown) ...
ок тестовый_запуск (tests.init.test_systemd.TestInitSystemSystemdRoNetworkd) ... ок тестовый_запуск

(tests.init.test_systemd.TestInitSystemSystemdRwFull) ... ок тестовый_запуск
(tests.init.test_systemd.TestInitSystemSystemdRwIfupdown) ... ок тестовый_запуск
(tests.init.test_systemd.TestInitSystemSystemdRwNetworkd) ... ок тестовый_запуск
(tests.init.test_busybox.TestInitSystemBusyboxRo) ... ок тестовый_запуск
(tests.init.test_busybox.TestInitSystemBusyboxRoNet) ... ок тестовый_запуск (tests.init.test_busybox.TestInitSystemBusyboxRw) ...
ок тестовый_запуск (tests.init.test_busybox.TestInitSystemBusyboxRwNet) ... ок
```

Выполнено 157 тестов за 0,021 с.

Хорошо

- Затем, чтобы запустить один тестовый случай:

```
$ support/testing/run-tests -d dl -o output_folder -k tests.init.test_busybox.  TestInitSystemBusyboxRw 15:03:26 TestInitSystemBusyboxRw
15:03:28 TestInitSystemBusyboxRw
15:08:18 TestInitSystemBusyboxRw 15:08:27
TestInitSystemBusyboxRw
.
.
.
Выполнено 1 тест за 301,140 с.
```

Начинается  
Задание  
Строительство завершено  
Уборка

Хорошо

Стандартный вывод показывает, был ли тест успешным или нет. По умолчанию вых однажды папка для теста удаляется автоматически, если не указана опция -k для сохранения одной директории.

### 22.7.1 Создание тестового случая

В репозитории Buildroot тестовая среда организована на верхнем уровне в support/testing/ по папкам conf, infra и tests. Все тестовые случаи находятся в папке tests и организованы в различных папках, представляющих категории теста.

Лучший способ познакомиться с тем, как создать тестовый случай, — это рассмотреть несколько базовых скриптов поддержки/тестирования/тестов/fs/ и init поддержки/тестирования/тестов/init/ файловой системы. Эти тесты дают хорошие примеры базовых тестов, которые включают как проверку результатов сборки, так и выполнение тестов во время выполнения. Существуют и другие, более сложные случаи, которые используют такие вещи, как вложенные папки br2-external для представления скелетов и дополнительных пакетов.

Создание базового тестового случая включает в себя:

- Определение тестового класса, который наследует от infra.basetest.BRTest
  - Определение члена конфигурации тестового класса, конфигурации Buildroot для сборки этого тестового случая. Он может опционально поддерживать флаги конфигурации, предоставляемые инфраструктурой тестирования времени выполнения infra.basetest.BASIC\_TOOLCHAIN\_CONFIG для получения базовой конфигурации архитектуры/цепочки инструментов и infra.basetest.MINIMAL\_CONFIG для того, чтобы не собирать никакую файловую систему. Преимущество использования infra.basetest.BASIC\_TOOLCHAIN\_CONFIG заключается в том, что предоставляется соответствующий образ ядра Linux, что позволяет загрузить полученный образ в Qemu без необходимости сборки образа ядра Linux в рамках тестового случая, что значительно сокращает время сборки, необязательное для тестового случая.
  - Реализация функции def test\_run(self): для реализации реальных тестов, которые будут запущены после завершения сборки. Это могут быть тесты, которые проверяют вывод сборки, запуск команд на хосте с помощью вспомогательной функции run\_cmd\_on\_host().
- Или они могут загружаться сгенерированную систему в Qemu, используя объект Emulator, доступный как self.emulator в тестовом примере. Например, self.emulator.boot() позволяет загружать систему в Qemu, self.emulator.login() позволяет войти в систему, self.emulator.run() позволяет запустить команды оболочки внутри Qemu.

После создания тестового случая добавьте себя в файл DEVELOPERS, чтобы стать ответственным за этот тестовый случай.

### 22.7.2 Отладка тестового случая

При запуске тестового случая папка output\_folder будет содержать следующие:

```
$ ls вых однажды_папка/
TestInitSystemBusyboxRw/
TestInitSystemBusyboxRw-build.log
TestInitSystemBusyboxRw-run.log
```

TestInitSystemBusyboxRw/ — это вых однажды каталог Buildroot, который сохраняется только в том случае, если передана опция -k.

TestInitSystemBusyboxRw-build.log — это журнал сборки Buildroot.

TestInitSystemBusyboxRw-run.log — это журнал загрузки и теста Qemu. Этот файл будет существовать только в том случае, если сборка прошла успешно и тестовый случай включает загрузку под Qemu.

Если вы хотите вручную запустить Qemu для проведения ручных тестов результата сборки, первые несколько строк TestInitSystemBusyboxRw-run.log содержат командную строку Qemu для использования.

Вы также можете внести изменения в текущие исходные коды внутри output\_folder (например, в целях отладки) и повторно запустить с стандартными целями Buildroot make (чтобы заново сгенерировать полный образ с новыми изменениями), а затем повторно запустить тест.

### 22.7.3 Тесты времени выполнения и Gitlab CI

Все тесты времени выполнения регулярно выполняются инфраструктурой Buildroot Gitlab CI, с м. .gitlab.yml и <https://gitlab.com/buildroot.org/-/buildroot/-/jobs>.

Вы также можете использовать Gitlab CI для тестирования новых тестовых случаев или проверки того, что существующие тесты продолжают работать после внесения изменений в Buildroot.

Для этого вам необходимо создать форк проекта Buildroot на Gitlab и иметь возможность отправлять ветви в свой форк Buildroot на Gitlab.

Имя отправляемой ветви определит, будет ли запущен конвейер Gitlab CI и для каких тестовых случаев.

В приведенных ниже примерах компонент <name> имени ветви предстает с собой произвольную строку, выбранную вами.

- Чтобы запустить все задания тестовых случаев run-test, отправьте ветку, которая заканчивается на -runtime-tests:

```
$ git push gitlab HEAD:<имя>-runtime-tests
```

- Чтобы запустить одно или несколько заданий тестового случая, отправьте ветку, которая заканчивается полным именем тестового случая (tests.init.test\_busybox.T) или с названием категории тестов (tests.init.test\_busybox):

```
$ git push gitlab HEAD:<имя>-<имя тестового случая>
```

Пример запуска одного теста:

```
$ git push gitlab HEAD:foo-tests.init.test_busybox.TestInitSystemBusyboxRo
```

Примеры запуска нескольких тестов, вх одних в одну группу:

```
$ git push gitlab HEAD:foo-tests.init.test_busybox $ git push gitlab HEAD:foo-tests.init
```

## Глава 23

# Файл DEVELOPERS и get-developers

Основной каталог Buildroot содержит файл DEVELOPERS, в котором перечислены разработчики, вовлеченные в различные области Buildroot. Благодаря этому файлу инструмент get-developers позволяет:

- Составить список разработчиков, которым следует отправить исправления, проанализировав исправления и опустив измененные файлы с соответствующими разработчиками. Подробности см. в разделе [22.5.](#)
- Найти разработчиков, которые занимаются данной архитектурой или пакетом, чтобы они могли быть уведомлены в случае сбоев сборки или ошибок на этой архитектуре или пакете. Это делается явно взаимодействии с инфраструктурой автосборки Buildroot.

Мы просим разработчиков, добавляющих новые пакеты, новые платы или вообще новые функции в Buildroot, зарегистрироваться в файле DEVELOPERS. Например, мы ожидаем, что разработчик, вносящий новый пакет, включит свой патч с соответствующей модификацией в файл DEVELOPERS.

Формат файла DEVELOPERS подробно документирован внутри самого файла.

Инструмент get-developers, расположенный в utils/, позволяет использовать файл DEVELOPERS для различных задач:

- При передаче одного или нескольких патчей в качестве аргумента командной строки get-developers вернет соответствующую команду git send-email. Если передана опция --cc, то будут выведены только адреса электронной почты в формате, подходящем для git send-email --cc-cmd.
- При использовании параметра командной строки -a <arch> get-developers вернет список разработчиков, отвечающих за данную архитектуру.
- При использовании параметра командной строки -p <package> get-developers вернет список разработчиков, ответственных за данный пакет.
- При использовании параметра командной строки -c get-developers просматривает все файлы под контролем версий в репозитории Buildroot и перечисляет те, которые не обрабатываются ни одним разработчиком. Цель этого параметра — помочь завершить файл DEVELOPERS.
- При использовании параметра командной строки -v проверяет есть ли в файле DEVELOPERS и выводит предупреждения для элементов, которые не совпадают.

# Глава 24

## Выпуски инженерных разработок

### 24.1 Выпуски

Проект Buildroot выпускает ежеквартальные релизы с ежемесчными исправлениями ошибок. Первый релиз каждого года — это релиз с долгосрочной поддержкой, LTS.

- Ежеквартальные выпуски: 2020.02, 2020.05, 2020.08 и 2020.11
- Выпуски с исправлениями ошибок: 2020.02.1, 2020.02.2, ...
- LTS-релизы: 2020.02, 2021.02, ...

Релизы поддерживаются до первого исправления ошибок с ледущего релиза, например, 2020.05.x прекращает свое существование после выпуска 2020.08.1.

Выпуски LTS поддерживаются до первого выпуска исправления ошибок LTS, например, 2020.02.x поддерживается до выпуска 2021.02.1.

### 24.2 Развитие

Каждый цикл выпуска состоит из двух месяцев разработки в ветке master и одного месяца стабилизации перед выпуском. В течение этой фазы в master не добавляются новые функции, только исправления ошибок.

Фаза стабилизации начинается тегом rc1, и каждую неделю до релиза тегируется еще один релиз-кандидат.

Для обработки новых функций и повышения версии во время фазы стабилизации может быть создана следующая ветка для этих функций. После того, как текущий релиз будет сделан, следующая ветка будет объединена с мастером, и цикл разработки следующего релиза продолжится.

## Часть 4

### Приложение

# Глава 25

## Документация по синтаксису Makedev

Синтаксис makedev используется в нескольких местах в Buildroot для определения изменений, которые необходимо внести в разрешения или для определения файлов устройств, которые необходимо удалить. create и как их создать, чтобы избежать вызовов mknode.

Этот синтаксис получен из утилиты makedev, более полную документацию можно найти в package/makedevs/README файл.

Он имеет форму списка полей, разделенных пробелами, по одному файлу на строку; поля следующие:

имя	тип	режим	uid	gid	главный	незначительный	начинать	вкл.	считать
-----	-----	-------	-----	-----	---------	----------------	----------	------	---------

Есть несколько нетривиальных блоков:

- имя — это путь к файлу, который вы хотите создать/изменить
- тип — тип файла, может быть одним из:
  - f: обычный файл, который должен уже существовать
  - F: обычный файл, который игнорируется при отсутствии
  - d: каталог, который создается вместе с его родителями, если отсутствуют
  - r: рекурсивный каталог, который должен уже существовать
  - c: файл символовического устройства родительский каталог которого должен существовать
  - b: блочный файл устройства родительский каталог которого должен существовать
  - p: названный канал, родительский каталог которого должен существовать
- режим — обычные настройки разрешений (допускаются только числовые значения); для типа d режим существующих родителей не изменен, но режим созданных родителей установлен; для типов f, F и r режим также может быть установлен в -1, чтобы не изменять режим (и только изменить uid и gid)
- uid и gid — это UID и GID, которые нужно задать для этого файла; могут быть как числовыми значениями, так и фактическими именами
- для файлов устройств здесь указаны основные и второстепенные, для симметричных файлов установлено значение -
- start, inc и count предназначены для создания пакета файлов и могут быть сведены к циклу, начинающемуся с start, увеличиваясь с счетчиком inc, пока он не достигнет count

Допустим, вы хотите изменить владельца и права доступа к определенному файлу. Используя этот синтаксис, вам нужно будет написать:

```
/usr/bin/foo f 755 0 0 -----
/usr/bin/bar f 755 корень корень -----
/data/buz f 644 бизнес-пользователь бизнес-группа -----
/data/baz f -1 baz-user baz-group -----
```

В качестве альтернативы, если вы хотите рекурсивно изменить владельца каталога, вы можете написать (чтобы установить UID на foo и GID на bar для каталога /usr/share/myapp и всех файлов и каталогов в нем):

```
/usr/share/myapp r -1 foo bar -----
```

С другой стороны, если вы хотите создать файл устройств /dev/hda и соответствующие 15 файлов для разделов, вам понадобится ядро /dev/hda:

```
/dev/hda b 640 корень корень 3 0 0 0 -
```

затем для файлов устройств, соответствующих разделам /dev/hda, /dev/hdaX, X в диапазоне от 1 до 15:

```
/dev/hda b 640 корень корень 3 1 1 1 1 5
```

Расширенные атрибуты поддерживаются, если включен параметр BR2\_ROOTFS\_DEVICE\_TABLE\_SUPPORTS\_EXTENDED\_ATTRIBUTES.

Это делается путем добавления строки, начинающейся с |xattr после строки, описывающей файл. В это же время в качестве расширенного атрибута поддерживается только capacity.

xattr	с помощью
-------	-----------

- |xattr — это «флаг», указывающий на расширенный атрибут
- возможность — это возможность добавить к предыдущему файлу

Если вы хотите добавить возможность cap\_sys\_admin в двоичный файл foo, вы напишете:

```
/usr/bin/foo f 755 root root ----- |xattr cap_sys_admin+eip
```

Вы можете добавить несколько возможностей в файл, используя несколько строк |xattr. Если вы хотите добавить возможности cap\_sys\_admin и cap\_net\_admin в двоичный файл foo, вы напишете:

```
/usr/bin/foo f 755 корень корень -----  
|xattr cap_sys_admin+eip |xattr  
cap_net_admin+eip
```

# Глава 26

## Документация по синтаксису Makeusers

Синтаксис создания пользователей вдохновлен синтаксисом makedev, приведенным выше, но специфичен для Buildroot.

Синтаксис для добавления пользователя предстает с собой список полей, разделенных пробелами, по одному пользователю на строку; поля следующие:

имя пользователя	uid	группа	гид	пароль домой	оболочка	группы	комментарий
------------------	-----	--------	-----	--------------	----------	--------	-------------

Где:

- имя пользователя — желаемое имя пользователя (также известное как имя в `/etc/passwd`) для пользователя. Оно не может быть `root` и должно быть уникальным. Если установлено значение `-`, то будет создан только группа.
- `uid` — желаемый UID для пользователя. Он должен быть уникальным, а не 0. Если установлено значение `-1` или `-2`, то уникальный UID будет вычислен Buildroot, где `-1` обозначает системный UID из диапазона [100...999], а `-2` обозначает пользовательский UID из диапазона [1000...1999].
- `group` — желаемое имя для новой группы пользователя. Оно не может быть `root`. Если группа не существует, она будет создана.
- `gid` — желаемый GID для новой группы пользователя. Он должен быть уникальным и не 0. Если установлено значение `-1` или `-2`, а группа не уже существует, то уникальный GID будет вычислен Buildroot, где `-1` обозначает системный GID из [100...999], а `-2` обозначающий GID пользователя из [1000...1999].
- `пароль` — это пароль, закодированный с помощью `crypt(3)`. Если префикс `!`, то вход в систему отключен. Если префикс `=`, то он интерпретируется как открытый текст, и будет зашифрован (используя MD5). Если префикс `!=`, то пароль будет зашифрован (используя MD5) и вход будет отключен. Если установлено значение `*`, то вход не разрешен. Если установлено значение `-`, то пароль не будет установлен.
- `home` — желаемый домашний каталог для пользователя. Если установлено значение `-`, домашний каталог не будет создан, а домашним каталогом пользователя будет `.`. Явное указание домашнего каталога `/` не допускается.
- `shell` — желаемая оболочка для пользователя. Если установлено значение `-`, то `/bin/false` устанавливается как оболочка пользователя.
- `groups` — это разделенный запятыми список дополнительных групп, к которым должен входить пользователь. Если установлено значение `-`, то пользователь будет членом без дополнительной группы. Отсутствующие группы будут созданы с произвольным gid.
- `комментарий` (он же [GECOS](#) поле) предстает с собой текст почти с любой формой.

На следующем примере показано, как поля накладываются друг на друга:

- запись ключением комментария в список полей оболочки для заполнения.
- поля записи ключением комментариев, не могут содержать пробелов.
- ни одно поле не может содержать двоеточие (`:`).

Если `home` не равен `-`, то домашний каталог и все файлы ниже будут принадлежать пользователю и его основной группе.

Примеры:

```
foo -1 bar -1 !=blabla /home/foo /bin/sh alpha,bravo Пользователь Foo
```

Это создаст следующего пользователя:

- имя пользователя (также известное как имя хода): foo
- uid вычисляется Buildroot
- основная группа bar
- основной групповой gid вычисляется Buildroot
- текстовый пароль: blabla, будет зашифрован с помощью crypt(3), вход в систему будет отключен.
- дом: /home/foo
- оболочка /bin/sh
- foo также является членом групп alpha и bravo
- комментарий: Foo пользователь

```
тест 8000 колесо -1 = - /bin/sh - Тестовый пользователь
```

Это создаст следующего пользователя:

- имя пользователя (также известное как имя хода): test
- идентификатор пользователя 8000
- основная группа колесо
- основной групповой gid вычисляется Buildroot и будет использовать значение, определенное в келете rootfs
- пароль пустой (иначе говоря пароль отсутствует).
- есть / но не будет принадлежать тесту
- оболочка /bin/sh
- тест не является членом каких-либо дополнительных групп
- комментарий: Тестовый пользователь

## 26.1 Предосторожение относительно автоматических UID и GID

При обновлении buildroot или при добавлении или удалении пакетов в/из конфигурации, возможно, что автоматические UID и GID будут изменены. Это может быть проблемой, если постоянные файлы были созданы с этим пользователем или группой: после обновления у них внезапно появится другой владелец.

Поэтому желательно увековечить автоматические идентификаторы. Это можно сделать, добавив таблицу пользователей сгенерированными идентификаторами. Это нужно делать только для UID, которые фактически ссылаются на постоянные файлы, например, базу данных.

## Глава 27

# Миграция с старых версий Buildroot

В некоторых версиях появлялись обратные несовместимости. В этом разделе объясняются эти несовместимости, и для каждой из них объясняется, что нужно сделать, чтобы завершить миграцию.

### 27.1 Общий подход

Чтобы перейти с старой версии Buildroot, выполните следующие действия.

1. Для всех ваших конфигураций выполните сборку в старой среде Buildroot. Запустите `make graph-size`. С остановите `graphs/file-size` в другом месте. Запустите `make clean`, чтобы удалить остальное.
2. Ознакомьтесь с конкретными примечаниями по миграции ниже и внесите необходимые изменения во внешние пакеты и пользовательские скрипты с борками.
3. Обновите Buildroot.
4. Запустите `make menuconfig`, начиная с существующего `.config`.
5. Если в меню Legacy что-либо включено, проверьте текст с правки, снимите флагок и с остановите конфигурацию.
6. Для получения более подробной информации просмотрите сообщения коммита git для нужных вам пакетов. Перейдите в каталог пакетов и выполните `git log <пакет>.. -- <ваш пакет>`.
7. Выполните сборку в новой среде Buildroot.
8. Исправление проблем с борками во внешних пакетах (обычно из-за обновленных зависимостей).
9. Запустите `make graph-size`.
10. Сравните новый файл `file-size-stats.csv` с исходным, чтобы проверить, не исчезли ли необходимые файлы и никаких новых больших ненужных файлов не появилось.
11. Для файлов конфигурации (и других) в пользовательском наложении, которые перезаписываются файлами, созданными Buildroot, проверьте, есть ли изменения в файле, созданном Buildroot, которые необходимо распространить на ваш пользовательский файл.

### 27.2 Переход на 2016.11

До Buildroot 2016.11 можно было использовать только одно дерево br2-external одновременно. С Buildroot 2016.11 появилась возможность использовать более одного одновременно (подробнее см. в разделе [9.2](#)).

Однако это означает, что старые внешние деревья br2 не могут использоваться как есть. Необходимо сделать небольшое изменение: добавить имя к внешнему дереву br2.

Это можно сделать очень легко, если озанес несколько шагов:

- Сначала создайте новый файл с именем external.desc в корне дерева br2-external с одной строкой, определяющей имя Ваше внешнее дерево br2:

```
$ echo 'имя ИМЯ_ВАШЕ Г О_ДРЕ ВА' >external.desc
```

Примечание. Будьте внимательны при выборе имени: оно должно быть уникальным и состоять только из символов ASCII из набора [A-Za-z0-9].

- Затем измените каждое вхождение BR2\_EXTERNAL в вашем дереве br2-external на новую переменную:

```
$ find . -type f | xargs sed -i 's/BR2_EXTERNAL/BR2_EXTERNAL_NAME_OF_YOUR_TREE_PATH/g'
```

Теперь ваше дерево br2-external можно использовать с Buildroot 2016.11 и более поздними версиями.

Примечание: это изменение делает ваше дерево br2-external не совместимым с Buildroot до версии 2016.11.

## 27.3 Переход на 2017.08

До версии Buildroot 2017.08 пакеты хрестасстанавливались в \$(HOST\_DIR)/usr (например, с помощью autotools --prefix=\$(HOST\_DIR)) в Buildroot 2017.08 они теперь устанавливаются непосредственно в \$(HOST\_DIR).

Всякий раз, когда пакет устанавливается полняемый файл, связанный с библиотекой в \$(HOST\_DIR)/lib, он должен иметь RPATH, указывающий на этот каталог.

RPATH, указывающий на \$(HOST\_DIR)/usr/lib, больше не принимается

## 27.4 Переход на 2023.11

До Buildroot 2023.11 бэкэнд загружки Subversion безоговорочно извлекал внешние ссылки (объекты со свойством svn:externals). Начиная с 2023.11 внешние ссылки больше не извлекаются по умолчанию; если они вам нужны, установите LIBFOO SVN\_EXTERNALS в значение YES. Это изменение подразумевает, что

- содержимое сконструированного архива может измениться, поэтому может потребоваться соответствующее обновление хешей;
- суффикс версии архива был обновлен до -br3, поэтому хеш-файлы должны быть обновлены соответствующим образом.

До Buildroot 2023.11 было возможно (но недокументировано и не использовалось) применять патчи, специфичные для архитектуры, добавляя к имени файла патча архитектуру в качестве префикса, например, 0001-some-changes.patch.arm, и такой патч применялся только для этой архитектуры. С Buildroot 2023.11 это больше не поддерживается, и такие патчи больше не применяются вообще.

Если вам по-прежнему нужны исправления для каждой архитектуры, вы можете предоставить предварительный хук, который копирует исправления, применимые к настроенной архитектуре, например:

```
определить LIBFOO_ARCH_PATCHES $  
  (foreach p,$(wildcard ${LIBFOO_PKGDIR}/*.patch.$(ARCH)), \  
   cp -f $(p) ${patsubst %.${(ARCH)},%,$(p)})  
 ) конец  
LIBFOO_PRE_PATCH_HOOKS += LIBFOO_ARCH_PATCHES
```

Обратите внимание, что ни один пакет в Buildroot не имеет патчей, специфичных для архитектуры, и такие патчи, содержащиеся в его исходном коде, не будут приняты.

## 27.5 Переход на 2024.05

Бэкэнды загрузки были расширены различными способами.

- Всё локально сгенерированные tarballs еще более воспроизводимы. До 2024.05 существовала возможность, что режим дос тупа к файлам в архивах не был с ограниченным, когда каталог загрузки имел определенные ACL (например, с набором ACL по умолчанию). Это влияет на архивы, сгенерированные для репозиториев git и subversion, а также для пакетов vendored cargo и go.
- Бэкэнд загрузки git теперь правильно расширяет **атрибут git export-subst** при создании архивов.
- Для создания архивов требуется новая версия tar, 1.35. В целях совместимости tar 1.35 изменяет способ хранения некоторых полей (devmajor и devminor), что влияет на метаданные, хранившиеся в архивах (но содержимое файлов не затрагивается).

Для учета этих изменений был обновлен или добавлен архивный суффикс:

- для git: -git4
- для subversion: -svn5
- для рузовых (ржавых) пакетов: -cargo2
- для пакетов go: -go2

Обратите внимание, что если два таких префикса будут применяться к сгенерированному архиву, например, для рузового пакета, загруженному с git, необходимо добавить оба суффикса сначала один для каждого из них, затем один для поставщика, например: libfoo-1.2.3-git4-cargo2.tar

Из-за этого эш-файл любых пользовательских пакетов или пользовательских версий для ядра и загрузчиков должен быть обновлен.

Следующие скрипты sed могут автоматизировать переименование в эш-файле (предполагая, что такие файлы уже есть в git):

```
# Для пакетов git и svn, которые изначально имели суффикс -br2 и соответственно -br3 sed -r -i -e 's/-br2/-git4/; s/-br3/-svn5/' $(git grep -l -E -- '-br2|-br3' -- '*.hash')
```

```
)
```

```
# Для пакетов go, которые изначально не имели суффикса sed -r -i -e 's/(\.tar\.gz)$/-go2\1/' $(
    git grep -l -E '\$(eval \$\((host-)?golang-package\))' -- '*.mk' | sed -r -e 's/\.mk$/.hash/' \
    | sort -u
)
```

```
# Для рузовых пакетов, которые изначально не имели суффикса sed -r -i -e 's/(\.tar\.gz)$/-cargo2\1/' $(
    git grep -l -E '\$(eval \$\((host-)?cargo- package\))' -- '*.mk' | sed -r -e 's/\.mk$/.hash/' \
    | sort -u
)
```

Обратите внимание, что эш изменится поэтому его также необходимо обновить (вручную).